to Gabriele and my family
Abstract

Web services are emerging as a promising technology for the development of next generation distributed heterogeneous software systems. Some prominent challenges for Web services are the discovery of services which match some given requirements, the composition of services into new complex applications, and the safe replacement of (sub)services.

The primary objective of this thesis is to define a suitable technique for a semantics- and behaviour-aware service discovery, capable of satisfying complex client requests which specify the ontology-annotated inputs and outputs, and (possibly) the expected behaviour of the service to be found.

We tackle the discovery, composition and replacement of services advertised by OWL-S descriptions, which provide a list of semantically annotated functional attributes of services, and a declaration of the interaction behaviour of services. First, we present a functional analysis that automatically generates (from a registry of OWL-S services) sets of services satisfying the functional requirements (viz., inputs and outputs) of client requests. Next, we describe a behavioural analysis that generates a (non-locking) composite service by suitably composing (the control-flow and the data-flow of) the services in a given set. The behavioural analysis checks whether the composite service satisfies the behavioural requirements of the query (viz., the expected service behaviour). To this end, a suitable notion of behavioural congruence for Web services is introduced.

In order to show the applicability in practice of such a discovery technique, we also present a proof-of-concept implementation of the functional and the behavioural analyses, and we discuss possible approaches to the issues of scalability and service heterogeneity.
Acknowledgments

I would like to especially thank my supervisor, Prof. Antonio Brogi, for his constant support, encouragement and professional guidance throughout my studies.

Special thanks go to Prof. Giuseppe Attardi and Dr. Vincenzo Gervasi for their feedback as internal reviewers of this thesis, and to Prof. Wolfgang Reisig and Dr. Massimo Paolucci for their feedback as external reviewers.

Last but not least, I would like to thank Prof. Fabio Gadducci and Filippo Bonchi for their support in modelling services, and Dr. Razvan Popescu, Fabrizio Benigni and Stefano Iardella for their cooperation in discovering services.
# Contents

1 Introduction  
1.1 Context and motivations .................................................. 11  
1.1.1 Web services’ overview ................................................. 11  
1.1.2 Some open issues in Web services .................................. 12  
1.2 Objectives of the thesis .................................................. 17  
1.3 Overview of the results .................................................. 19  
1.3.1 Functional analysis .................................................... 19  
1.3.2 Behavioural analysis ................................................... 20  
1.3.3 The discovery framework ............................................. 21  
1.4 Structure of the thesis .................................................. 24  

2 Background and state of the art  
2.1 Background: OWL-S and Petri nets .................................... 27  
2.1.1 OWL-S, a semantic markup for Web services ..................... 27  
2.1.2 An example: five OWL-S process models .......................... 30  
2.1.3 A short introduction to Petri nets .................................. 35  
2.2 State of the art ......................................................... 36  
2.2.1 Single service discovery .............................................. 37  
2.2.2 Composition-oriented service discovery ............................ 39  
2.2.3 Service composition ................................................... 41  
2.2.4 Service replaceability .................................................. 42  
2.2.5 Our previous results .................................................. 43  
2.2.6 Discussion ............................................................. 47  

3 Functional analysis  
3.1 Collecting functional information ...................................... 50  
3.1.1 The dependency hypergraph ......................................... 51  
3.1.2 Constructing the dependency hypergraph ........................... 57  
3.1.3 An example ............................................................ 60  
3.2 Satisfying the functional requirements of a client query ........... 65  
3.2.1 Discovering sets of services ........................................ 67  
3.2.2 Discovering minimal sets of services ............................... 68  
3.2.3 Soundness, completeness and complexity ........................... 70
3.2.4 An example ........................................ 74
3.3 Discussion ........................................... 78

4 Behavioural analysis 81
4.1 Modelling service behaviour with Petri nets ................. 82
  4.1.1 CPR nets: Consume- Produce- Read nets ............... 83
  4.1.2 From OWL-S descriptions to Petri nets .............. 85
  4.1.3 OCPR nets: Open Consume-Produce-Read nets ....... 86
  4.1.4 A compositional encoding for OWL-S ............... 88
  4.1.5 An example .................................. 91
4.2 From sets of services to compositions of services ............ 94
  4.2.1 Constructing the composite service ................. 95
  4.2.2 An example .................................. 96
4.3 Reasoning on service behaviour ................................ 98
  4.3.1 A behavioural congruence for Web services .......... 99
  4.3.2 Checking behavioural properties of Web services ... 105
  4.3.3 An example .................................. 115
4.4 Related work on nets and net equivalences ................. 119
4.5 Discussion ........................................... 122

5 SAM: a proof-of-concept implementation 127
5.1 An architecture overview ................................ 127
5.2 Some implementation details ................................ 130
5.3 SAM in action ....................................... 133
  5.3.1 Web interface ................................ 133
  5.3.2 Submitting a service ........................... 134
  5.3.3 Submitting a functional query ...................... 135
5.4 Experimental evaluation ................................ 136
5.5 Discussion ........................................... 138

6 Concluding remarks 141
6.1 Summary ........................................... 141
6.2 On the applicability of the proposed techniques .......... 144
  6.2.1 On the scalability of service discovery .......... 145
  6.2.2 Dealing with other service description languages ... 151

Bibliography 157

A Proof of Theorem 4.1 167
List of Figures

1.1 A book-selling service. ........................................ 15
1.2 A transaction service. ........................................ 16
1.3 Thesis’ placement w.r.t. discovery, composition, invocation. .... 19
1.4 A *bird’s eye view* of the discovery framework. ............... 21

2.1 A sample atomic process in the presentation syntax. ............ 29
2.2 A sample composite process in the presentation syntax. ......... 30
2.3 A *tree-view* of the CreditPortal process model. ................ 32
2.4 A *tree-view* of the RatingOne process model. .................. 33
2.5 A *tree-view* of the RatingTwo process model. .................. 34
2.6 A *tree-view* of the HotelService process model. ............... 34
2.7 A *tree-view* of the ConferenceService process model. ......... 35

3.1 A directed hyperedge. ......................................... 51
3.2 Pseudo-code for computing the *profiles* of a service. ........ 55
3.3 Pseudo-code for computing the alternative executions of a process. 56
3.4 Pseudo-code for building the dependency hypergraph. .......... 59
3.5 A view of the hypergraph after inserting the *e-commerce* ontology. 60
3.6 A view of the hypergraph after inserting the *event* ontology. .... 61
3.7 A view of the hypergraph after inserting the ConferenceService. .... 62
3.8 A view of the hypergraph after inserting the hotel ontology. ....... 62
3.9 A view of the hypergraph after inserting the HotelService. ....... 63
3.10 A view of the hypergraph after inserting the bankOntology. .... 64
3.11 A view of the hypergraph after inserting the RatingOne service. .... 65
3.12 A view of the hypergraph after inserting the RatingTwo service. .... 66
3.13 Pseudo-code summarising the functional analysis. ............... 67
3.14 Example of *minimal* sets of services. .......................... 69
3.15 An OWL-S process model representing a *behavioural* client query. .... 77

4.1 Modelling atomic operations as CPR net transitions. ............ 84
4.2 Modelling OWL-S composite operations as CPR nets. ............ 86
4.3 Two open nets, a context and a composite net. .................. 87
4.4 Context and extended context of the choice operator. ........... 89
4.5 Contexts of the OWL-S operators extended for \( A = \{ A_1, \ldots, A_n \} \). .... 90
4.6 OCPR net representation of the atomic processes of HotelService.  ... 91
4.7 OCPR net representation of the HotelService (Step 1). ............... 91
4.8 OCPR net representation of the HotelService (Step 2). ............... 92
4.9 OCPR net representation of the HotelService. ......................... 92
4.10 OCPR net representation of the RatingOne service (Step 1). ....... 93
4.11 OCPR net representation of the RatingOne service (Step 2). ...... 93
4.12 OCPR net representation of the RatingOne service (Step 3). ....... 94
4.13 OCPR net representation of the ConferenceService. ................. 97
4.14 Parallel composition of HotelService and ConferenceService. ....... 98
4.15 OCPR net representation of the RatingTwo service. ................. 99
4.16 Example of (non-)equivalent services. ............................... 101
4.17 Two sample OCPR nets. ............................................. 106
4.18 The LTS derived from the firing semantics of the net $N_1$. ......... 107
4.19 Pseudo-code for deriving a LTS from an OCPR net. .................. 108
4.20 Pseudo-code for the lock analysis. .................................. 110
4.21 The LTS derived from the firing semantics of the net $N_1$. ......... 114
4.22 The LTS derived from the firing semantics of the net $N_2$. ......... 114
4.23 The product of the LTSs derived from nets $N_1$ and $N_2$. ........ 115
4.24 The OCPR net representation of the client query. .................... 116
4.25 Public specification of the CreditPortal service. .................... 117
4.26 OCPR net representation of the CreditPortal service. ............... 118
4.27 OCPR net representation of the CreditPortal public specification. ... 118
4.28 The OCPR net 1. ...................................................... 121
4.29 Two similar (according to [48]) OCPR nets that are saturated bisimilar. 122

5.1 The global architecture of SAM. ..................................... 128
5.2 SAM implementation. .................................................. 132
5.3 The Web interface of SAM. ........................................... 133
5.4 Submitting the HotelService to SAM. ................................ 135
5.5 Submitting a functional query to SAM. ................................ 136
5.6 The results of a functional query. .................................... 137
5.7 A comparison between SAM and OWLS-MX. .......................... 139
5.8 Average query response time of SAM. ................................ 139

6.1 Indexing textual documents. ......................................... 146
6.2 Indexing compositions of services. .................................. 148
6.3 A minimalist architecture for indexing services. .................... 150
6.4 The discovery framework embedded in a UDDI registry. ............. 156

A.1 The context $ADD_A^O$ and $SUB_B^O$, for interface $O = (i, f, \{A, B\})$. 170
Chapter 1

Introduction

The first Section of this Chapter overviews the area of Web services (Subsection 1.1.1) and briefly discusses some of their current limitations (Subsection 1.1.2). Next, Sections 1.2 and 1.3 provide a bird’s-eye view of the main objectives and contributions of this thesis, while the overall structure of the thesis is described in Section 1.4.

1.1 Context and motivations

The Web is rapidly evolving from being a collection of static information to a collection of services which interoperate through the Internet. Recently, increasing attention is devoted to service-oriented computing [68], a new emerging paradigm for distributed computing whose best-known instantiation is represented by Web services.

1.1.1 Web services’ overview

Service-oriented computing [68] is emerging as a new promising computing paradigm that centres on the notion of service as the fundamental element for developing software applications. Quoting the W3C consortium, a Web service is “a software application identified by a URI, whose interfaces and bindings are capable of being defined, described and discovered by XML artefacts. A Web service supports direct interactions with other software agents, using XML-based messages exchanged via Internet-based protocols” [90].

The Web service architecture [89] defines three component roles: service provider, service requester and service broker. A service provider sends a description of the featured service to a service broker, which publishes such an information in a service repository. The service broker manages the repository and allows service requesters to find services. A service requester sends a description of the desired service (i.e., a

---

1In this thesis, we use the terms “service” and “Web service” interchangeably.
query) to the service broker, which returns the services matching the request. Then, the requester interacts with the respective service.

The Extended Markup Language (XML) [91] provides a common, platform-neutral syntax for defining both Web services’ standards as well as messages exchanged between services, hence supporting interoperability and platform-neutral communications.

The three current standards for Web services are the Simple Object Access Protocol (SOAP), the Web Service Description Language (WSDL), and the Universal Description Discovery and Integration (UDDI).

SOAP [88] is a lightweight protocol for the exchange of information in a decentralized, distributed environment. The key element of a SOAP message is the envelop item, which can wrap any kind of XML-message exchanged between services. SOAP is language, operating system, platform and, also, transport-protocol independent.

WSDL [93] is an XML-based language for describing Web services and how to access them. A WSDL 1.1 document describes a Web service using five major elements: ports, which specify the Internet locations where a service can be accessed, and port types, messages, types and binding, which define the operations performed, the messages, the data types and the communication protocols used by the service. A client program connecting to a Web service can read its WSDL interface to determine which operations are available, then it can use SOAP messages to actually call one (or more) of such operations, thus abstracting from platform-dependent and implementation details.

UDDI [82] is a platform-independent, XML-based registry for service descriptions where providers register themselves together with the services they offer, and from where requesters find such information. UDDI is designed to be queried by SOAP messages and to provide access to WSDL documents describing the protocol information and message formats required to interact with the registered Web services. A UDDI service registration consists of three components: white pages, i.e., address, contact, and known identifiers of the provider, yellow pages, i.e., industrial categorisations based on standard taxonomies such as NAICS (North American Industry Classification System) or UNSPSC (Universal Standard Products and Services Classification), green pages, i.e., technical information about the service. UDDI employs such information to feature taxonomy-based (viz., “yellow pages search”) and keyword-based (viz., “white/green pages search”) searches.

1.1.2 Some open issues in Web services

Web services are emerging as a promising technology for the development of next generation distributed heterogeneous software systems [2]. Three of the open prominent challenges for the advancement of Web services are:

- Web service discovery, to find and re-use existing services,
• **Web service composition**, to rapidly build new complex applications,

• **Web service replaceability**, to safely substitute (sub)services.

Web service discovery refers to the ability of finding the “right” service, namely, a service which matches some given requirements. Recently, the number of services available is rapidly increasing, so that manually browsing existing registries to locate a desired service is really a too expensive task. A major advantage that a developer can obtain from an effective service discovery mechanism is to reduce application development time and cost, since a rapid location of services promotes the re-use of existing software components.

Web service composition is the task of (automatically) generating a new value-added service by combining (parts of) existing services. A first advantage of service composition is the possibility to accelerate the development of new complex applications, thus avoiding to re-implement already existing components. Another benefit of composing services is the possibility to satisfy complex client requests. An immediate example of this is a client wishing to plan its holidays by booking flight or train tickets as well as hotel accommodation while taking into account various parameters such as weather, season prices, special offers, and so on. A service developer can satisfy the client request by creating a *Travel Agency* application, which suitably interacts, for instance, with a *Flight Reservation* service, a *Train Reservation* service, a *Hotel Booking* service and a *Weather Forecast* service.

Web service replaceability is the ability of substituting a service with another one, in such a way that the change is transparent to the (clients of the) applications interacting with the replaced service. In other words, a service taking part in an application can be replaced by another service, if the latter does not alter the (externally observable) behaviour of the application. Note that Web service replaceability also refers to the ability of safely substituting a part of a service (viz., a subservice), not altering the behaviour of the entire service. A major advantage of the ability of replacing (sub)services is to provide the stability of service-oriented applications. It is often the case that a service becomes unexpectedly unavailable: for example, the provider does not offer the service any more, or simply, the provider releases a new version (which adds/removes some features) of the service.

Currently, standard WSDL interfaces provide services with purely syntactic descriptions. A first issue toward a high-quality service discovery is the lack of machine-interpretable, semantic information: any possibility of reasoning on service capabilities is inhibited. This is the case of UDDI, which features only keyword-based or taxonomy-based syntactic matches. For instance, a car-selling service which may be self-described as a *Car Dealer* does not match a request for an *Automobile Dealer*. A syntactic search for the keyword “flute” may return a *Musical Instruments Seller* as well as a *Wine Glasses Seller*, one of which is probably not suitable for satisfying the request.
A promising solution emerges from the Semantic Web [13] initiative, which aims at bringing semantics to the Web. Roughly, the Semantic Web provides a common framework for representing and publishing on the Web machine-interpretable structured information. A key element is the notion of ontology, which asserts relationships among entities hierarchically organised into classes [51]. A semantic Web service [52] employs ontologies to self-describe in a machine-interpretable way its functional (i.e., inputs and outputs) and non-functional (e.g., service category and quality of service) properties. In the last years, several approaches have emerged to enhance WSDL service descriptions with semantics information, such as SAWSDL [30], OWL-S [64], WSMO [94] or SWSO [80]. Semantic Web services enable high-quality semantics searches (e.g., [66, 41, 5, 44]) which advance keyword-based searches with reasoning on services’ capabilities, thus not failing due to syntactic mismatches and false positive matches. For example, if ontology information is employed, an Automobile Dealer service matches a request for a Car Dealer, since automobile and car are synonyms. A SUV Dealer service matches a request for a Car Dealer as well, since Car subsumes SUV, viz., SUV is more specific than car. The word flute is disambiguated, since it belongs to two different ontologies, one describing the musical instruments’ domain and the other modelling the glasses’ domain. A semantic search for a service capability flute of the musical instruments’ (wine glasses’) ontology returns the Musical Instruments Seller (Wine Glasses Seller) only.

It is often the case that a client query cannot be fulfilled by a single service, while it may be fulfilled by a suitable composition of services. Currently, WSDL interfaces do not include any protocol information, that is, the order in which the operations are to be invoked. As a consequence, composing services only on the basis of WSDL interfaces – even if semantically enhanced – may lead to composite services that fail or lock (e.g., [5, 8]). For example, the interaction with a Photo Printer service that exposes a WSDL interface defining the operations login, sendPhotos and payment fails if the sendPhotos operation is invoked either before login or after payment. Furthermore, a composition may run into a situation where the interaction behaviour of two services exclude each other, for example, the former waits for a message from the latter and vice versa.

Several approaches have emerged to enhance service descriptions with behaviour information. A service is not simply represented as a collection of isolated WSDL operations. A behaviour interface (e.g., [19, 92, 64, 94]) describes a service as a structured flow of interactions with other services. Low level process modelling languages [19, 92] support (human) software developers that, however, have to manually compose (the control and data flows of) services. Automation of information use and dynamic data interoperability are the primary objectives of Semantic Web Services [64, 94, 80], which enable automatic service discovery and composition. For example, a (complex) request for planning holidays can be satisfied by automatically finding and composing suitable services (e.g., Flight Reservation, Train Reservation and Hotel Booking).
Protocol information is obviously needed to satisfy requests which ask for services with specific behaviour [65, 1]. This is the case of a software developer that needs a service to complete an application.

For example, consider the BookSellingService illustrated in Figure 1.1. Such a service consists of the parallel (viz., split+join) composition of two (sub-)services, namely the OrderManager and the PaymentService. The OrderManager is a sequential (viz., sequence) composition of the chooseBook atomic process and of a conditional choice (viz., if-then-else) between the atomic processes finaliseOrder and returnMessage. Similarly, PaymentService is a sequential composition of the CC_Checker atomic process and of a conditional choice between the atomic processes executeTransaction and invalidCard.

The usual behaviour of the BookSellingService is the following. When a client provides an ISBNCode, the chooseBook atomic process returns the price of the selected book. Then, the CC_Checker inputs the book price and the client credit card data (viz., cc_Number, cc_ExpirationDate, cc_Owner), it checks whether the client provided a valid credit card, and it returns a validationCheck parameter. If the client credit card is valid, the finaliseOrder atomic process provides some necessary details (viz., transactionDetails) to complete the selling transaction, otherwise the returnMessage atomic process returns an error message to the client. Next, if the client credit card is valid, the executeTransaction atomic process performs the selling transaction and returns an invoice, otherwise the invalidCard atomic process returns an abortMessage.

Let us now suppose that the BookSellingService provider wants to substitute the PaymentService (e.g., the BookSellingService provider may search for a low-cost service).
payment service). In such a case, the provider of the BookSellingService needs a new service to safely substitute PaymentService (viz., the dotted area in Figure 1.1), so to continue to offer the book-selling service.

A crucial issue is to check whether such a new service and the existing part of the application will interact properly, for instance, without locks and without pending messages which cannot be received any more [69, 45]. For instance, the service may contain internal choices relevant for a correct behaviour of the application that, yet, is not informed of which decision is actually made.

The BookSellingService provider may search for a service which employs the functional attributes of the PaymentService, hence formulating a query which specifies price, cc.Number, ccExpirationDate, cc.Owner, validationCheck and transactionDetails as input, and validationCheck, invoice and abortMessage as output. For instance, the TransactionService illustrated in Figure 1.2 matches such a functional query.

![Figure 1.2: A transaction service.](image)

Yet, the interaction of the OrderManager service and the TransactionService fails. Indeed, after executing the chooseBook atomic process, the OrderManager locks, since both the atomic processes finaliseOrder and returnMessage need the validationCheck parameter to execute. On the other hand, the TransactionService locks as well, since the executeTransaction atomic process needs the transactionDetails parameter – to be provided by the (locked) finaliseOrder process – to execute.

We hence argue that functional information does not suffice to address the service replaceability issue. Protocol information is needed to formulate behavioural queries which ask for services featuring a specific behaviour. For example, the BookSellingService provider may search for a service which behaves as the PaymentService (viz., the (sub-)service to be replaced), hence not matching the TransactionService in Figure 1.2.

Another prominent problem emerges from the heterogeneity of service descriptions. A service description can provide different types of information (i.e., signature only, semantics, behaviour, or both semantics and behaviour) and each type can be described by means of different description languages (e.g., WSDL [93], SAWSDL [30], WS-BPEL [19], OWL-S [64]). Most service description languages are built on top of WSDL, yet, if semantics and/or behaviour of the services to be found/composed/replaced are described employing different semantics/protocol
languages, it is not possible to discover services, to compose them as well as to check their interaction protocol in an automatic way. Service heterogeneity has promoted the development of tools for the automated translation of service protocols (e.g., [4, 67]). However, they are in their early stages, since they feature only partial translations, thus providing only a partial contribution to the issues of service heterogeneity.

The open issues of (automatic) service discovery, composition and replacement have been disjointedly tackled, so far. Various approaches to semantics service discovery are now available (e.g., [44, 66, 79, 7]), some of them also coping with (simple) requirements on service behaviour [1, 65], yet, none of such proposals cope with complex requests that need of combining the capabilities of several services. Dually, approaches defining suitable techniques to compose a given set of services [81, 72, 11] do not consider how such a set can be determined. Several replaceability relations on services [69, 74, 28, 42] have been recently proposed, however, although they provide suitable theoretical foundations to service replaceability, they do not take into account how to select (compositions of) services candidate to replace a given service. Service discovery, composition and replaceability require further investigations and a general, well-founded methodology to a composition-oriented discovery is still missing. The most recent proposals on discovery, composition and replaceability of services will be overviewed in Section 2.2.

1.2 Objectives of the thesis

The two main objectives of this thesis are:

1. to define a suitable technique for a composition-oriented service discovery capable of exploiting both ontology and behaviour information available in descriptions of services,

2. to define a discovery technique which can be applicable in practice.

We argue that ontology information is needed to feature high-quality, automatic discovery processes, matching services’ outputs and services’ inputs in spite of (syntactic) differences. Behaviour information is needed to generate service compositions, in order to solve complex client requests which require to compose the functionalities provided by several services, and to analysis service behaviour, to check whether a (composite) service may lock, as well as whether a (composite) service features a required behaviour.

A behaviour- and ontology-aware service discovery provides an important contribution to the previously mentioned open issues of Web service composition and Web service replaceability. Indeed, a behaviour- and ontology-aware discovery may automatically find and suitably compose the services necessary to achieve a desired
service. Furthermore, a behaviour- and ontology-aware discovery may find a service (composition) which is behaviourally equivalent to a desired service.

We hence consider service descriptions enhanced with ontology and behaviour information, and in particular, we consider OWL-S service descriptions. OWL-S [64] is a computer-interpretable semantic mark-up language, where ontology-based descriptions of service capability and of interaction service behaviour coexist. We chose OWL-S because: (i) it was the first (and it is hence the more mature) initiative to include both ontology and behaviour information in service descriptions, (ii) it relies on OWL [51], which is the current standard to describe ontologies, (iii) it provides a (simple) notation for building up composite services in terms of control and data flow (differently, e.g., from WSMO [94] which focuses on specifying service choreographies).

Two main issues toward the definition of an applicable ontology- and behaviour-aware service discovery technique are:

- the availability of heterogeneous service descriptions,
- the high computational cost needed to analyse ontology and behaviour information.

The available services are mostly described by means of syntactic WSDL interfaces, while only a few of them provide ontology and/or behaviour information. Recently, WS-BPEL [19] has been approved as the OASIS standard for expressing Web service compositions, while several initiatives such as [30, 54] and [72] aim at adding semantics to standard WSDL and WS-BPEL, respectively. Such a heterogeneity of service descriptions determines two important (sub-)objectives of the thesis, namely:

(3) to define a common formalism to represent (ontology and behaviour) information available in heterogeneous service descriptions,

(4) to design a discovery framework that: (a) provides a suitable translation mechanism to express (OWL-S) service descriptions into the common formalism, (b) manages a local registry which stores the translated services, and (c) accesses the local registry to satisfy the client requests.

In particular, we employed Petri nets to express behaviour information, and we defined a suitable data structure where ontology information is summarised by means of dependencies among ontology concepts.

There are several reasons why we chose Petri nets to model service behaviour. Petri nets [76] are one of the best known and most widely adopted formalisms to express the concurrent behaviour of (software) systems. Besides providing a clear and precise semantics, they feature an intuitive graphical notation, and a number of techniques and tools for their analysis, simulation and execution are available [71]. Furthermore, Petri nets have also been already employed to model Web services
1.3. OVERVIEW OF THE RESULTS

(e.g., see [34, 47, 83, 77]), and there are translators available (like [63, 38]) capable of converting “non-OWL-S” service descriptions into Petri nets. This suggests the adoption of Petri nets as a *lingua franca* for expressing and for reasoning about heterogeneous service descriptions.

Efficiency and scalability are two main requisites of any search mechanisms. Unfortunately, if on the one hand, a behaviour- and ontology-aware discovery may guarantee high-quality results, on the other hand it is a time consuming task. Hence, a further (sub-)objective of the thesis is to show the feasibility of the proposed discovery technique, and in particular:

(5) to implement a proof-of-concept prototype of the discovery framework, and

(6) to discuss the adaptability of the indexing techniques developed for keyword-based search engines to an ontology and behaviour-based service discovery.

Finally, Figure 1.3 illustrates the placement of this thesis with respect to the traditional split of tasks service discovery first, service composition second and service invocation third. The thesis – which presents a technique for a composition-oriented discovery – clearly addresses the task of service discovery, however, as we will better describe in Chapters 3 and 4, it partially contributes to the task of service composition as well, since the ability of combining functionalities of different services.

Figure 1.3: Thesis’ placement w.r.t. discovery, composition, invocation.

1.3 Overview of the results

In this Section we overview the main contributions of this thesis. In particular, two suitable techniques to analyse ontology and protocol information of services are briefly discussed in Subsections 1.3.1 and 1.3.2, respectively. A discovery framework for registering and discovering services is next introduced in Subsection 1.3.3.

Preliminary versions of the results presented in this thesis have been published in [27, 21, 10].

1.3.1 Functional analysis

Given a registry of (OWL-S) services and a client request specifying the ontology-annotated inputs and outputs of the service to be found (viz., a so-called *functional* query), the functional analysis presented in this thesis (viz., in Chapter 3) automatically generates sets of services that satisfy the client request. The functional analysis
first analyses the (OWL-S) descriptions of the available services in order to determine dependencies between service inputs and outputs, and next, it collects such dependencies into a hypergraph. Intuitively, nodes correspond to ontology concepts, while hyperedges represent dependencies among them. Note that a dependency can represent equivalence relations between concepts defined in separate ontologies, as well as, it can link the inputs to the outputs of a service, to state that such outputs can be generated if such inputs are available. The hypergraph is then explored to determine sets of services which satisfy the given client request. Roughly, a set of services satisfies (the functional requirements of) a client request if such services generate the requested outputs, and if they take as input (a sub-set of) the inputs of the client request.

An interesting feature of the functional analysis is the ability of considering, if necessary, *multiple* executions of the same service. For example, if a *choice* service which inputs a parameter of type `city` and generates either a parameter of type `weatherForecast` or a parameter of type `temperature` is available, a client query specifying `city` as input and `weatherForecast` and `temperature` as outputs can be satisfied by suitable invoking such a service twice. Moreover, it is important to note that only *minimal* set of services are returned. Roughly, a set of services is minimal if each of such services is strictly necessary to satisfy the query. For instance, let us suppose that an atomic service which inputs `city` and generates `weatherForecast` and `temperature` is now available. A set of services which contain both the *choice* service and the atomic service is *non* minimal, since the atomic service can satisfy the client query by itself.

### 1.3.2 Behavioural analysis

Given a set of services which satisfies (the functional requirements of) a client request, the behavioural analysis presented in this thesis (viz., Chapter 4) first suitably composes (the control-flow and the data-flow of) the services in the set in order to generate a composite service. Then, it provides two different behavioural analyses, according to the type of the client query to be satisfied:

1. **lock analysis** – which is performed when a *functional* query (viz., a query specifying ontology-annotated inputs and outputs of the desired service) is received. The lock analysis checks whether the composite service satisfies the request (viz., it generates the requested outputs) without locking.

2. **bisimilarity analysis** – which is performed when a *behavioural* query (viz., a query specifying ontology-annotated inputs and outputs, and the expected behaviour of the desired service) is received. The bisimilarity analysis checks whether the composite service behaves according to the client request.

The behavioural analysis employs OCPR nets [16] – a variant of standard Condition/Event Petri nets – to model service behaviour. Briefly, an OCPR net (for Open
1.3. OVERVIEW OF THE RESULTS

Consume-Produce-Read net) is equipped with two disjoint sets of places, namely, control and data places, to naturally model the control flow and the data flow of a Web service, and with an interface, which establishes those data places that can be observed externally. Then, a suitable notion of behavioural equivalence for Web services expressed as OCPR nets is defined. Three properties of such an equivalence are weakness, as it equates externally indistinguishable services by abstracting from the number of internal steps, compositionality, as it is also a congruence, and computability, as the set of states that an OCPR net can reach is finite. Given a (composite) service and the behaviour description of the requested service, the bisimilarity analysis employs such a notion of behavioural equivalence to check whether the given service and the requested service feature the same (externally observable) behaviour.

1.3.3 The discovery framework

As one can observe in Figure 1.4, functional and behavioural analyses are the two key components of the discovery framework presented in this thesis. Roughly, the framework provides two main functionalities which consist of registering new services and searching for services.

![Figure 1.4: A bird’s eye view of the discovery framework.](image)

Registering new services

As previously anticipated, an internal representation of services is defined. In particular, a service is expressed in terms of nodes and hyperedges by the functional analysis, while it is modelled as an OCPR net by the behavioural analysis. When a request for adding a new (OWL-S) service to the registry is received, the framework (i.e., the functional and behavioural analyses, respectively) parses the OWL-S
service description in order to build a hypergraph portion as well as an OCPR net which model the new service, and it adds them to the registry. In such a way, the time needed to translate services into the internal representation does not affect the query answering time. On the other hand, we expect that requests for searching services will occur more frequently than the ones for registering new services.

**Searching for services**

As depicted in Figure 1.4, clients can search for services by submitting to the discovery framework two different types of queries, namely, *functional* queries and *behavioural* queries. A *functional* query is a query which specifies (ontology-annotated) inputs and outputs of the service to be found. In this case, the discovery framework returns those (non-locking compositions of) services which are capable of generating all the query outputs starting from (a sub-set of) the query inputs. Yet, in some cases, a client may need to search for a service which features a specific behaviour. For example, such a client may search for a service to be plugged-in to a complex application, so that the new service has to behave accordingly to the interaction behaviour of the existing application. A *behavioural* query is a query which specifies inputs, outputs and the desired behaviour of the service to be found. The discovery framework hence returns those (compositions) of services which satisfy the functional requirements (i.e., inputs and outputs) of the query by featuring also a query-equivalent behaviour. Note that a client can easily formulate a behavioural query by providing the OWL-S description of the desired service, which is then translated into the internal service representation (in particular, into OCPR nets) by the discovery framework.

Let us now overview the behaviour of the whole discovery framework, which is sketched in Figure 1.4. When a functional query is received, functional and behavioural analyses cooperate in order to determine all the (compositions of) services which satisfy the given query without locking. First, the functional analysis explores the hypergraph – which collects the functional dependencies among the available services – and it determines the minimal sets of services which satisfy the query, if any. No behavioural property is guaranteed, yet. Next, each set is analysed by the behavioural analysis, which performs a *lock analysis* on the composite service resulting from the parallel composition of the services in the set. Intuitively, the lock analysis checks whether (1) whether there exists a terminating behaviour of the composite service which generates the requested outputs, or (2) whether all the possible behaviour of the composite service terminate and generate the requested outputs. The composite services which pass the termination check comply the functional query and they constitute the final output of the discovery framework for this type of query.

When a behavioural query is received, it is splitted in two parts, the first consisting of the inputs and the outputs, and the second describing the desired behaviour of the service to be found. Roughly, a functional query is a sub-set of a behavioural query. Accordingly, the functional part of the query is solved firstly, by generating
the composite services capable of satisfying such a query. Next, given a composite service the behavioural analysis performs a *bisimilarity analysis*, which checks whether a service features a query-equivalent behaviour. The output of the discovery framework when receiving a behavioural query consists of the composite services which pass the bisimulation check.

**Main features**

We complete the overview of the discovery framework by providing a short discussion of its main features.

1. **Ability of addressing functional and behavioural queries.**
   The discovery framework is capable to tackle both *functional* and *behavioural* queries. Given a functional query, i.e., a query specifying the functional attributes of the desired service, the (compositions of) services which satisfy the query without locking are returned. When receiving a behavioural query, i.e., a query specifying the functional attributes and the behaviour of the desired service, the framework returns the (compositions of) services which satisfy the query and which feature the requested behaviour. Those services are guaranteed to be used interchangeably with the service described by the behavioural query. The discovery framework hence addresses the emerging problem of service replaceability.

2. **Composition-oriented matching.**
   The discovery framework is able to generate suitable service compositions, so to satisfy queries which cannot be fulfilled by a single service. The services in the composition are guaranteed to interact properly.

3. **Ontology-based matching.**
   The discovery framework is capable of performing flexible matching between service inputs and outputs automatically, so to establish functional dependencies among services. An output of a service is matched with an input of another service if the service output is described by an ontology concept *equivalent to* or *more specific than* the ontology concept which describes the service input. Functional and behavioural queries are hence automatically processed by the discovery framework, which does not require human interaction.

4. **Pre-processing of the query-independent tasks.**
   The discovery framework has been designed to pre-compute the query-independent tasks, such as, the translation of (the descriptions of) services into the internal representation (i.e., hypergraph and OCPR nets) and the setting of functional dependencies among services, thus avoiding to affect the query answering time.
1.4 Structure of the thesis

The two main contributions of the thesis, viz., functional analysis and behavioural analysis are described in Chapters 3 and 4, respectively, while a proof-of-concept implementation of the proposed discovery technique is presented in Chapter 5. In order to make the thesis self-contained, we include a short introduction to OWL-S and to Petri nets in Chapter 2. Moreover, state-of-the-art and related work is discussed in Chapter 2, and separately analysed at the end of Chapters 3 and 4.

Chapter 2 briefly introduces the key elements of OWL-S ontologies and Petri nets in Section 2.1. Next, it discusses the mostly related state-of-the-art approaches on service discovery, composition and replaceability in Section 2.2.

Chapter 3 describes the analysis of functional information of services. Section 3.1 first formally introduces the dependency hypergraph (Subsection 3.1.1) and then illustrates how to construct it (Subsection 3.1.2). The dependency hypergraph collects semantic relationships among ontology concepts and functional dependencies among the inputs and the outputs of services. We present a sample dependency hypergraph in Subsection 3.1.3. Section 3.2 describes how to select (sets of) services which satisfy the functional requirements (viz., inputs and outputs of the desired service) of a given client request. In particular, Subsection 3.2.1 presents a discovery algorithm which explores the dependency hypergraph in order to determine all the sets of services capable of satisfying the functional requirements of a client request, and Subsection 3.2.2 shows that such sets are minimal, that is, they contain only services strictly necessary to satisfy the request. Soundness, completeness and complexity of the algorithm presented in Subsection 3.2.1 are discussed in Subsection 3.2.3. An example of the behaviour of the discovery algorithm in Subsection 3.2.1 is presented in Subsection 3.2.4. Finally, we present some concluding remarks in Section 3.3.

Chapter 4 describes the analysis of behavioural information of services. Section 4.1 discusses how to employ Petri nets to model service behaviour. In particular, we introduce Consume- Produce-Read nets (CPR nets for short) in Subsection 4.1.1 and we present a first informal encoding from OWL-S descriptions to CPR nets in Subsection 4.1.2. In Subsection 4.1.3, we extend CPR nets with a notion of net interface (viz., a set of externally observable open places), so to define the Open Consume- Produce-Read nets (OCPR nets for short). A formal and compositional encoding from OWL-S descriptions to OCPR nets is introduced in Subsection 4.1.4, and an example of such an encoding is illustrated in Subsection 4.1.5. Next, we discuss how to generate composite services in Section 4.2, where we describe how to compose OCPR nets (Subsection 4.2.1) and where we present a suitable example (Subsection 4.2.2). Section 4.3 describes how to establish whether a (composition of) service(s) satisfies the behavioural requirements (viz., the expected behaviour of the desired service) of a given client request. In particular, Subsection 4.3.1 defines a
suitable notion of behavioural congruence for Web services modelled as OCPR nets, Subsection 4.3.2 presents and discusses suitable algorithms for verifying whether a (composite) service terminates and features a behaviour equivalent to the client requirements, and Section 4.3.3 illustrates an example. Finally, related work on nets and net equivalences is discussed in Section 4.4, while some concluding remarks are drawn in Section 4.5.

Chapter 5 presents a proof-of-concept prototype – named SAM (for Service Aggregation Matchmaking) – which implements the discovery technique introduced in Chapters 3 and 4. The architecture, some implementation details, a friendly Web interface and some experimental results of the proof-of-concept prototype are described in Sections 5.1, 5.2, 5.3 and 5.4, respectively. Some concluding remarks are drawn in Section 5.5.

Chapter 6 briefly summarises the main contributions of the thesis in Section 6.1 and draws some final concluding remarks on the applicability in practice of the proposed techniques in Section 6.2.
Chapter 2

Background and state of the art

This Chapter provides a brief introduction to the key building bricks of this thesis, viz., OWL-S ontologies and Petri nets (Section 2.1), and a discussion of the more closely related state-of-the-art approaches to the discovery, composition and replaceability of Web services (Section 2.2).

2.1 Background: OWL-S and Petri nets

In order to make the thesis self-contained, in this Section we provide an introduction to OWL-S and to Petri nets. In particular, the three main parts of OWL-S, i.e., service profile, process model and service grounding are briefly analysed in Subsection 2.1.1, some sample OWL-S services are presented in Subsection 2.1.2, while the basic notions of Petri nets are introduced in Subsection 2.1.3.

2.1.1 OWL-S, a semantic markup for Web services

The OWL-based Web Service ontology (OWL-S, [64]) has been designed by the OWL-S coalition to enable the automation of the discovery, composition and invocation of Web services. “OWL-S is a OWL-based Web service ontology, which supplies Web service providers with a core set of markup language constructs for describing the properties and capabilities of their Web services in unambiguous, computer-interpretable form” [64].

The OWL-S ontology has three primary sub-ontologies: the service profile, the process model and the grounding, each of them modelling a different view of a service. The service profile describes what the service does, the process model describes how the service is used, and the service grounding describe how to interact with the service. Each OWL-S service description consists of a (instance of the class) process model, of one (instance of the class) grounding or more, and, optionally, of one (instance of the class) profile or more. While the process model provides an abstract description of how to interact with the service, the grounding complements
the process model providing protocol and message format information to access the
service. OWL-S allows a service to expose multiple profiles, each of them providing
a different high-level service description suitable for service selection.

An OWL-S service profile defines three aspects of a service: (1) the \textit{functional aspect},
that is, inputs, outputs, preconditions and effects, (2) the \textit{classification aspect},
that is, the type of service as specified in a business taxonomy, and (3) the \textit{non-
functional aspect}, that is, a variety of service parameters which specify additional
features of the service, such as, security and privacy requirements, precision, cost,
response time, availability, quality of service. A crucial aspect is that the values of
the types of OWL-S inputs and outputs are URIs (Uniform Resource Identifiers)
of concepts defined in some ontology. A major advantage of Web-addressable on-
tology concepts is the enabling of discovery processes capable of avoiding syntactic
mismatches of service parameters.

The OWL-S process model describes how the service performs its component
tasks. More precisely, the process model describes a service as a composite process
which consists, in turn, of composite processes and/or atomic processes. An atomic
process can not be decomposed further (viz., it has no internal structure) and it
executes in a single step (similarly to a \textit{black box} providing a functionality). In
particular, an atomic process is defined by its inputs, outputs, preconditions and
results\footnote{Although OWL-S permits to specify also preconditions and results of service executions, we
decided to consider their possible employment neither within client requests nor within the discov-
er-process, as their specification/usage is not quite clear.}.

Figure 2.1 illustrates a sample atomic process, viz., the \texttt{signIn} atomic process
of the \texttt{HotelService} (Figure 2.6). \texttt{signIn} takes as input the \texttt{contactInfo} parameter,
which is an instance of the \texttt{contactInformation} concept defined in the \texttt{e-commerce}
ontology, and it produces as output the \texttt{fee} parameter, which is an instance of the
\texttt{registrationFee} concept of the \texttt{e-commerce} ontology, and the parameters \texttt{city}, \texttt{country},
\texttt{initDate}, \texttt{endDate}, which are instances of the concepts \texttt{city}, \texttt{country}, \texttt{startDate}
and \texttt{endDate}, respectively, defined in the \texttt{event} ontology. Note that Figure 2.1 provides a
more readable \textit{presentation syntax} \cite{49} of the \texttt{signIn} definition, rather than showing
the (non human-oriented) canonical RDF/XML syntax.

A composite process consists of a set of component processes and it is built up by
using a few control constructs: \texttt{sequence} (i.e., sequential execution), \texttt{if-then-else}
(conditional execution), \texttt{choice} (non-deterministic execution), \texttt{split} (parallel execution), \texttt{split+join}
(parallel execution with synchronization), \texttt{any-order} (unordered sequential execution), \texttt{repeat-while}
and \texttt{repeat-until} (iterative execution). More precisely \cite{64}:

\begin{itemize}
  \item a \texttt{sequence} process is a list of processes to be executed in order;
  \item a \texttt{choice} process is a bag of processes out of which only one can be chosen for
  execution;
\end{itemize}
with_namespaces
(uri"http://www.di.unipi.it/~corfini/.../HotelService.owl",
e-commerce: uri"http://www.di.unipi.it/~corfini/.../e-commerce.owl",
ext: uri"http://www.di.unipi.it/~corfini/.../event.owl",
process: uri"http://www.daml.org/services/owl-s/1.2/Process.owl",
rdf: uri"http://www.w3.org/1999/02/22-rdf-syntax-ns",
...

{ (define atomic process signIn
  (inputs: (contactInfo – e-commerce:contactInformation)
    preconditions: ...
    outputs: (fee – e-commerce:registrationFee
      city – event:city
      country – event:country
      initDate – event:startDate
      endDate – event:endDate)
    results: ...
    )
  ...
}

Figure 2.1: A sample atomic process in the presentation syntax.

- a split process is a bag of processes to be executed concurrently;
- a split+join process is a bag of processes to be executed concurrently with barrier synchronisation;
- an if-then-else process is a bag of two processes out of which one is chosen for execution according to the value of a condition;
- an any-order process is a bag of processes to be executed in some unspecified order but not concurrently;
- a repeat-while process is a process to be executed zero or more times, until a condition becomes false;
- a repeat-until process is a process to be executed at least once, until a condition becomes true.

Figure 2.2 illustrates (the presentation syntax of) the sequence#14 composite process, taking part in the CreditPortal service (Figure 2.3). The sequence#14 process takes as input two parameters, viz., amountOfCredit and guarantee, which are instances of the concepts amountOfCredit and guarantee respectively, defined in the bankOntology. Furthermore, sequence#14 is a sequence composite process, which performs the sequential execution of the processes changeAmountOfCredit and
changeGuarantee. The parameters newCredit and newGuarantee of the component processes correspond to the input parameters amountOfCredit and guarantee of the sequence#14 composite process.

```{with_namespaces
(uri"http://www.di.unipi.it/~corfini/.../CreditPortal.owl",
 bankOntology: uri"http://www.di.unipi.it/~corfini/.../bankOntology.owl",
 ...}

(def define composite process sequence#14
  (inputs: (amountOfCredit – bankOntology:amountOfCredit
             guarantee – bankOntology:guarantee)
  preconditions: ...
  results: ...
  perform changeAmountOfCredit(newCredit ← amountOfCredit);
  perform changeGuarantee(newGuarantee ← guarantee);
  ...)

Figure 2.2: A sample composite process in the presentation syntax.

The OWL-S grounding specifies how to concretely organise inputs and outputs of atomic processes as messages. Roughly, OWL-S atomic processes correspond to WSDL operations, while inputs and outputs of OWL-S atomic processes correspond to (part of) WSDL messages. Namely, the grounding provides a bridge between the high-level and abstract OWL-S descriptions and the real WSDL interfaces.

In the following Chapters we concentrate on the OWL-S process model, since it provides essential information for discovering and composing services, which is the main objective of this thesis. Indeed, even if the information included in the service profile would be sufficient for (single) Web service discovery, it would not suffice for composing services. In particular, the profile-based service discovery/composition does not suffice in the case of complex services, while in the case of atomic services the information provided by the profile coincides with the information advertised by the process model. As previously described, the service profile provides a list of the inputs and outputs of a service, yet it provides no information about the order in which the inputs are requested, or the outputs are produced. Hence, we argue that it is not possible to compose services in a (semi-)automatic engineered way using their profile contents only.

2.1.2 An example: five OWL-S process models

This Subsection is devoted to present the OWL-S descriptions of five sample services that we will use in the following chapters to illustrate the behaviour of the discovery
framework. A detailed discussion of the OWL-S process model of each service is included, while we discard the presentation of the service profile and grounding, since the framework does not need them to discover and generate composite services.

An OWL-S process model is a complex XML document describing a service in terms of its constituent processes. For the sake of simplicity, we present a more compact, readable tree-view of an OWL-S process model, rather than listing the actual OWL-S code\(^2\). In particular, as one can observe, for example, in Figure 2.3, each internal node (i.e., a grey node) represents a composite process, whose type (e.g., sequence, choice, and so on) is denoted by a label. By convention, we identify an internal node with the string controlConstruct\(\#N\). For example, sequence\(\#1\) denotes the root (composite) process of the process model illustrated in Figure 2.3. Furthermore, in case of conditional (viz., if-then-else) and iterative control constructs (viz., repeat-until and repeat-while), a condition is associated to the corresponding internal node. Note, for instance, the condition \((\text{rejectedLogin} = \text{true})\) associated to the node if-then-else\(\#4\) in Figure 2.3. Each leaf node (i.e., a white node), instead, represents an atomic process, whose name is specified by the label of the corresponding node. Moreover, a leaf has associated the input and output ontology concepts of the represented atomic process. For example, note in Figure 2.3, the inputs \(\text{username}, \text{password}\) and the outputs \(\text{validData}, \text{rejectedLogin}\) of the atomic process \(\text{login}\). Each of them is a concept defined in some ontology. In this case, as stated by the legend of Figure 2.3, they belong to the \text{bankOntology}. Furthermore, we use the italic style to identify those ontology concepts with boolean primitive type.

We can now introduce the OWL-S process models of the services \text{CreditPortal}, \text{RatingOne} and \text{RatingTwo}, operating in the bank domain, and of the services \text{HotelService} and \text{ConferenceService}, operating in the domains of tourism and academic, respectively.

\textbf{The CreditPortal service}

\text{CreditPortal} is a Web service that grants loans to bank customers. It implements three steps, namely, (1) authentication of the customer and upload of her/his personal data, (2) evaluation of the customer credit request, and (3) formulation of the loan offer. Briefly, in the first step, after logging into the bank system, the customer has to upload information regarding balance and offered guarantees. In the second step, \text{CreditPortal} evaluates the customer reliability and computes a rating of the credit request. Finally, in the last step \text{CreditPortal} either decides to grant the loan to the customer and to build an offer, or it rejects the credit request.

The tree-view of the \text{CreditPortal} process model is depicted in Figure 2.3. \text{CreditPortal} consists of a sequence process whose left-most child is a repeat-until construct representing the customer authentication phase. The customer can repeatedly

\footnote{\text{The full OWL-S code of the process models introduced in this Subsection is available at http://www.di.unipi.it/~corfini/owls/owls.html.}}
choose between logging in with an existing account (login) or creating a new account (createAccount) until either the log in to the system is successful (validData = true), or the system rejects the login definitively (rejectedLogin = true). Next, if the customer did not provide a valid login, CreditPortal terminates (invalidLogin). Otherwise, it continues by repeatedly asking the customer for the personal financial data (validateClientData) until either a valid information is uploaded (validateResponse = true) or the system rejects the credit request (changeClientData = false). Then, if the customer did not provide valid financial data, CreditPortal terminates (rejectClientData), otherwise the customer credit request evaluation phase starts. CreditPortal, taking into account the requested amount of credit, firstly evaluates the customer security (securityEvaluation), computes the customer rating (computingRating) and next decides whether or not to make an offer to the customer (makeOffer). If so (makeOffer = true), it builds the offer (buildOffer), formally confirms the offer (confirmOffer) and asks the customer for a final confirmation (userConfirmation). Next, if CreditPortal and the customer agree on the offer (confirmation = true and userConfirmation = true), the offer is finalized (finalizeCredit), otherwise CreditPortal rejects the
2.1. BACKGROUND: OWL-S AND PETRI NETS

credit request \( (\text{rejectResponse}) \). Instead, if \text{CreditPortal} does not want to make an offer \( (\text{makeOffer} = \text{false}) \), it can choose either to reject the credit request \( (\text{rejectCredit}) \) or to allow the customer to update the financial data \( (\text{changeAmountOfCredit} \text{ and } \text{changeGuarantee}) \). In the latter case, the customer credit request evaluation phase is repeated.

The RatingOne service

\text{RatingOne} is a service which evaluates customer rating. Given some sensitive customer data, such as the requested amount of credit, the balance and the provided guarantees, \text{RatingOne} first performs three different evaluations of the customer rating. Next, it computes the final rating by averaging the intermediary evaluations.

Figure 2.4 provides the tree-view of the \text{RatingOne} process model. \text{RatingOne} consists of the sequential composition of a \text{split+join} composite process and of an atomic process. Accordingly to different criteria, three distinct evaluations of the customer rating are jointly performed \( (\text{computeFirstRating}, \text{computeSecondRating}, \text{computeThirdRating}) \). Then, the final (average) rating is computed \( (\text{averageRating}) \).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{rating_one.png}
\caption{A tree-view of the \text{RatingOne} process model.}
\end{figure}

The RatingTwo service

\text{RatingTwo} is a service which evaluates customer rating, as well. Yet, differently from \text{RatingOne}, it computes the customer rating and only if necessary (e.g., if the first rating exceeds a threshold value) it performs a second and possibly a third evaluation of the customer.

The tree-view of the \text{RatingTwo} process model is illustrated in Figure 2.5. \text{RatingTwo} is a \text{sequence} process composed, in turn, of an atomic process and of an \text{if-then-else} composite process. Given the customer data, i.e., the requested amount of credit, the balance and the guarantees, \text{RatingTwo} firstly evaluates the customer rating \( (\text{computeFirstRating}) \). If such a rating is lower than some threshold value \( (\text{firstRating} < x) \), then the final rating is computed \( (\text{computeFinalRating}) \). Otherwise, a second evaluation of the customer rating is performed \( (\text{computeSecondRating}) \). Again, if the second rating evaluation is lower than some threshold value \( (\text{secondRating} < x) \), then the final rating is computed \( (\text{averageRating}) \) by averaging
the first and the second rating values. Otherwise, a third evaluation of the rating is also computed (computeThirdRating). Next, the first, second and third costumer ratings are further analysed to establish the final rating (evaluateRating).

```plaintext
Figure 2.5: A tree-view of the RatingTwo process model.
```

The Hotel service

The HotelService allows a client to search for and/or to reserve hotels. The treeview of the HotelService process model is depicted in Figure 2.6. HotelService is a repeat-until of a choice between two processes. Indeed, a client can repeatedly choose between searching for a hotel (collectInfoHotel) and completing a hotel reservation. In the latter case, a specific hotel has to be first selected (selectHotel), next, the client can finalize the hotel reservation with the payment (payment). As one may observe in Figure 2.6, the functional attributes of HotelService are annotated with the concepts of two separate ontologies, namely, the hotel and e-commerce ontologies.

```plaintext
Figure 2.6: A tree-view of the HotelService process model.
```
2.1. BACKGROUND: OWL-S AND PETRI NETS

The Conference service

ConferenceService allows a client to register to academic events. Figure 2.7 depicts the tree-view of the ConferenceService process model. ConferenceService consists of the sequential execution of three processes, respectively corresponding to the following steps: selection of an academic event (selectEvent), submission of client personal data to complete the registration (signIn), and payment of the registration fee by means of either credit card (payWithCreditCard) or bank transfer (payWithBankTransfer). As emphasised in Figure 2.7, concepts of both the event and e-commerce ontologies are employed to annotate the functional attributes of ConferenceService.

![Figure 2.7: A tree-view of the ConferenceService process model.](image)

2.1.3 A short introduction to Petri nets

Petri nets were introduced by Carl A. Petri [70] for modelling concurrent behaviour of a distributed system. A crucial element for the Petri net theory is the notion of net [76].

**Definition 2.1 (nets).** A net is a triple \( N = (P, T, F) \) where:

1. \( P = \{p_1, p_2, ..., p_m\} \) is a finite set of places,
2. \( T = \{t_1, t_2, ..., t_n\} \) is a finite set of transitions,
3. \( F \subseteq (P \times T) \cup (T \times P) \) is a set of arcs (i.e., the flow relation),
4. \( P \cap T = \emptyset \) and \( P \cup T \neq \emptyset \).

A net is a directed and bipartite graph whose nodes can be distinguished in two non-empty and disjoint sets (4), named places (1) and transitions (2). Places are connected to transitions as well as transitions are connected to places by means of directed (3) arcs. Hence, an arc can connect only two (differently typed) nodes. By convention, places and transitions are graphically represented by circles (or ellipses) and rectangles, respectively.

For each transition \( t \in T \), we define the pre-set and the post-set of \( t \).
Definition 2.2 (pre-, post-set). Given a net $N = \langle P, T, F \rangle$, for each $t \in T$ the sets
\[
\diamond t = \{ p \in P \mid (p, t) \in F \} \quad t^\circ = \{ p \in P \mid (t, p) \in F \}
\]
denote the pre-set and post-set of $t$, respectively.

Places can contain objects, standardly named tokens. A marking is an assignment of tokens to places. It represents the state of the net, that changes whenever tokens modify their distribution. The dynamic behaviour of a net is given by transition firings, which consume and produce tokens.

Depending on the number of tokens that can be assigned to a place and on the possible information carried by a token, different classes of Petri nets are defined.

Condition/Event nets (C/E nets) are Petri nets where tokens are just markers for places and every place contains at most one token [76]. A marking $M$ for a C/E net $N$ is hence a finite set of places (i.e., $M \subseteq P$).

Definition 2.3 (C/E firing step). Let $N = \langle P, T, F \rangle$ be a C/E net. Given a transition $t \in T$ and a marking $M$ for $N$, a firing step is a triple $M[t]M'$ such that

1. $\diamond t \subseteq M$,
2. $t^\circ \cap M = \emptyset$, and
3. $M' = M \setminus \diamond t \cup t^\circ$.

Given a C/E net $N$ and a marking $M$ for $N$, a transition $t \in T$ is enabled if (1) all conditions required by $t$ hold ($\diamond t \subseteq M$), and (2) if no conditions ensured by $t$ hold ($t^\circ \cap M = \emptyset$). An enabled transition $t$ in a marking $M$ can fire. Then, the new marking $M' = M \setminus \diamond t \cup t^\circ$ is reached (3).

2.2 State of the art

In this Section we briefly discuss (some of) the most recent proposals on discovery, composition and replaceability of services. We first observe that such proposals can be classified with respect to two orthogonal criteria. For example, we can distinguish semantics-based approaches, behaviour-based approaches and semantics- and behaviour-based approaches, depending on whether they analyse semantics information (viz., ontology-based descriptions of the functional attributes of services), behavioural information (viz., descriptions of the protocol interaction of services) or both, respectively. Another possible classification criterion – applied in this Section – is to categorise approaches with respect to the issue(s) which they address. In particular, we identified four main issues, each of them denoting a class of approaches: (1) single service discovery, (2) composition-oriented service discovery, (3) service composition and (4) service replaceability.
The rest of this Section is organised as follows. Subsections 2.2.1, 2.2.2, 2.2.3 and 2.2.4 are devoted to briefly discuss (some of) the most recently proposed approaches to single service discovery, composition-oriented discovery, service composition and service replaceability, respectively. We summarise our previous proposals to a composition-oriented discovery in Subsection 2.2.5. Finally, Subsection 2.2.6 provides a high-level comparison of the works presented in Section 2.2. To simplify the reading, we delay a more focussed discussion of the (comparative) advantages of the approach described in this thesis until Sections 3.3 and 4.5, when the key components of our discovery technique will have been introduced.

2.2.1 Single service discovery

In this Subsection we briefly discuss (some of)\(^3\) the approaches addressing the issue of a single service discovery. We consider first the capability-based approaches, which select services with respect to the functionalities they provide.

In particular, the capabilities of services can be described in two different ways, the first providing a class hierarchy where each class denotes a set of similar services, and the second providing a description of the service functionalities and their transformations. The *RatingOne* service in Figure 2.4, for example, belongs to the class of the financial services, yet, it can be also described as a service which inputs the balance, the provided guarantee and the requested amount of credit of a client and which produces as output a rating evaluation. Concerning the classification-based approaches, it is worth mentioning the work of Li and Horrocks. They described in \([44]\) a service discovery algorithm based on a description logic reasoner, which speeds up the matching process by employing an (off-line) classification of DAML-S (the predecessor of OWL-S) service advertisements. Note that the search mechanisms supported by UDDI belong to the classification-based approaches, as well.

An explicit description of the capabilities of services is taken into account by the approach of Paolucci et al. which proposed in \([66]\) the first matchmaking algorithm based on the DAML-S service profile. Their algorithm takes as input a client query, a repository of DAML-S Web services, as well as the shared type ontology, and searches for services able to satisfy the query. The client specifies the query as a list of provided inputs and requested outputs. An advertisement matches the request \([66]\) if all the outputs of the request match outputs of the advertisement, and dually, all the inputs of the advertisement match inputs of the request. The matched services have associated a degree of match: exact (when the inputs/outputs of the advertisement are equivalent to the inputs/outputs of the request), plug-in (when the inputs/outputs of the advertisement include the inputs/outputs of the request), or subsumes (when the inputs/outputs of the request include the inputs/outputs of

---

\(^3\)In particular, we do not describe here the discovery techniques based on UDDI only, since the discovery approach presented in this thesis considers enhanced service descriptions providing both semantics and behaviour information.
the advertisement). [66] sorts matched services by selecting first the match with the highest score in the outputs. Input matching is used only as a secondary score to tie breaks between equally scoring outputs.

It is interesting to note that the algorithm in [66] has been also employed to enhance the search functionality of UDDI. In particular, Srinivasan et al. [79] first defined a mapping of OWL-S service profiles into UDDI service representations, then they suitably extended the UDDI registry interface as well as the UDDI API to integrate the OWL-S capability matching into the UDDI registry (as we will better discuss in Section 6.2.2).

The semantic matching of service capabilities defined in [66] has been extended in [41], where Klush et al. introduced a service matchmaker employing both logic based reasoning and IR techniques to relate inputs/outputs of a service with the input/output of a client request. In particular, besides the (logic-based) exact, plug-in and subsumes matches defined in [66], two additional hybrid matches, viz., the (logic- and IR-based) subsumed-by match and nearest-neighbor match, are introduced. The subsumed-by match selects services whose output data is more general than the requested output data, and whose syntactic similarity with the request is above a given threshold. Similarly, the nearest-neighbor match selects services whose output data is more specific of the requested output data, and whose syntactic similarity with the request is above a given threshold.

Bansal and Vidal [7] were the first to propose a service discovery algorithm that takes into account service behaviour. Their algorithm analyses DAML-S process models (rather than service profiles as, e.g., [66, 41, 5]). Similarly to [66], their algorithm takes as input a query specifying the desired inputs and outputs as well as a repository of DAML-S Web services, and returns one of the following degrees of match: exact, plug-in, subsumes, or failed. A service request matches a service advertisement if the request provides all the inputs (possibly more) needed by the advertisement while the advertisement generates all the outputs (possibly more) needed by the requester. In particular, the algorithm defined in [7] initially executes on the root composite process of a DAML-S process model, and next, it recursively visits all the child (composite/atomic) processes finishing at the atomic processes. For each composite process (e.g., sequence, choice, and so on) as well as for atomic processes a corresponding matching algorithm is employed. For example, in the case of a sequence process, if the outputs requested by the query can be satisfied by all its children collectively then we have a success, otherwise a failure.

The approaches described so far have been designed to satisfy requirements on the capabilities of services. An approach to a process-level service discovery has been recently proposed by Traverso et al. in [65]. Given a set of candidate services (previously selected by means of traditional capability-level discovery techniques like [66]), the approach in [65] searches for a service whose process-level description satisfies some given client requirements on the behaviour of the desired service. In particular, they consider a global ontology defining the relevant concepts of the discovery
scenario, a set of annotated (abstract) BPEL processes [72] modelled as annotated state transition systems, and some discovery requirements expressed by means of (CTL) temporal logic. Then, they employ (existing) model checking techniques to determine whether a (annotated) state transition system satisfies the discovery (behavioural) requirements.

Requirements on the behaviour of services are taken into account also by the approach of Agarwad and Studer [1]. First, they define a new formalism, which mix together description login and \(\pi\)-calculus, to model semantic, temporal and security constraints of services. Client request are espressed by means of the same formalism. Then, they defined a matchmaking algorithm that automatically returns the services which match the given semantic, temporal and security requirements.

### 2.2.2 Composition-oriented service discovery

Aversano et al. and Benatallah et al. respectively proposed in [5] and [8] two approaches which extend [66] with the discovery of service compositions.

The algorithm of Aversano et al. [5] analyses DAML-S service profiles (as [66]) and, by performing a cross-ontology matching (over service descriptions employing different ontologies), it searches for service compositions capable of satisfying the client request (when no single service can fulfill the request). The algorithm starts by searching for those services which generate the requested outputs. Then, it recursively continues by searching for those services producing the unavailable inputs (viz., the inputs provided neither by the query nor by the matched services, if any) of the previously matched services. In case that the request can be satisfied, the algorithm returns all service combinations that can satisfy it.

The algorithm of Benetallah et al. [8] computes the combinations of Web services that best match a given request by resolving a best covering problem in the domain of hypergraphs theory. Each available service becomes a vertex in the hypergraph while each query output \(O_Q\) becomes an edge populated by those services that produce an output equivalent to \(O_Q\). The problem of computing the best service combinations can be reduced to the computation of the minimal transversals (i.e., covers) with the minimal cost of the hypergraph, where the notion of cost is defined in terms of the missing information of the request with respect to the considered service combination. It is worth noting that whilst [5] consider the possibility that the missing input of a service (i.e., the inputs not contained in the client request) could be produced as output by other services, [8] do not take into account this feature. It is also worth observing that, being based on the analysis of service profiles, [5, 8] do not consider service behaviour, thus returning sets of services which may satisfy the request, since they possibly lock during their interaction.

Another approach to a composition-oriented service discovery was proposed by Ben Mokhtar et al. in [57]. The algorithm in [57] models (OWL-S described) services as well as the client request as finite state automata, and it then tries to reconstruct
the client query automaton by using fragments of the services’ automata. Such an algorithm performs in two main steps. First, it selects a set of services that may be useful to reconstruct the query automaton, by employing the semantic matching of [66]. Next, it generates a global automaton by suitably connecting the automata of the filtered services, and it checks whether there exists (at least) a sub-automaton of the global one that behaves as the query automaton. If so, the algorithm returns a list of the selected (sub-)automata. Yet, a major limitation of [57] is a too strict match between the services’ automata and the query automaton. For example, [57] does not match a service that first takes \( A \) and produces \( C \), and then takes \( C \) and produces \( B \) with a query that takes \( A \) and directly produces \( B \).

A similar work has been presented by Hashemian and Mavaddat in [36], where they introduced a graph-based approach for composing (OWL-S described) services. They formally model services as well as the client request as interface automata [29]. In particular, a service (as well as query) is denoted as a triple \( \langle A_{Iws}, A_{Ows}, k_{ws} \rangle \), where \( A_{Iws} \), \( A_{Ows} \) are the inputs and outputs of the service \( ws \) respectively, while \( k_{ws} \) represents the set of dependencies between the inputs and outputs of \( ws \). Such triples are then synthetised in a graph, where each node represents an input or an output of some available service. There is an edge connecting two nodes \( x, y \), if a dependency between \( x \) and \( y \) is contained in (at least) a dependency set of some available service. Given a query graph, the algorithm in [36] checks whether there exists a sub-graph of the global graph that covers all the edges which belong to the query graph. It is worth noting, that [36] analyses input/output dependencies among services, yet, it does not consider the ordering of atomic processes (within services) which is crucial in checking the correct behaviour of a service composition (e.g., to determine whether the composition may lock).

In [61], Oh et al. defined the composition-oriented discovery problem as an AI planning problem. First, they introduced their notion of flexible parameter matching, namely, two parameters \( p_1, p_2 \) match if \( p_1.type = p_2.type \) or \( p_1.type \) is derived from \( p_2.type \) in a type hierarchy (viz., in an ontology). Given a request specifying the sets \( I, O \) of the desired input and output parameters, the composition-oriented discovery problem consists of finding a finite sequence of services \( w_1, w_2, \ldots, w_n \) such that \( w_i \) can be invoked sequentially after executing \( w_1, \ldots, w_{i-1} \), with \( i \in [2, n] \), and the requested outputs \( O \) are satisfied by the outputs of \( w_1, w_2, \ldots, w_n \) (and the invocation costs of \( w_1, w_2, \ldots, w_n \) is minimized). Next, such a problem is redefined as a AI planning problem in a state spaces \( \Psi = \langle S, s_0, s_G, \Omega(.) \rangle \), where each state \( s \in S \) is a collection of parameters, \( s_0 \) is the initial state, such that \( s_0 = I \), \( s_G \) is the goal state, such that \( s_G \subseteq O \), and each state \( s \in S \) is associated with the services \( \Omega(s) \) which can be invoked in \( s \). It is worth noting that only sequential composition of service is considered.

Our previous works [27, 21] on a composition-oriented service discovery will be presented in Subsection 2.2.5.
2.2. STATE OF THE ART

2.2.3 Service composition

Benatallah and Hamadi defined in [34] a Petri net based algebra for modelling Web service control flows. They use ordinary Petri nets to represent services and their compositions. A service is denoted by a single transition and two places for absorbing and emitting information, respectively. Then, the authors provide several compositional operators, such as, e.g., sequence, alternative, iteration, parallel with communication, selection and refinement, to construct service compositions. In particular, they define a direct mapping from each of such operators to a Petri net construction.

An interesting approach to an automated composition of (semantic) Web services was proposed by Traverso and Pistore in [81]. They illustrated how OWL-S process models can be automatically composed to generate new composite services which can be next deployed and executed on engines for process execution languages, like BPEL [19]. In particular, given the OWL-S process models of n available services $W_1, W_2, \ldots, W_n$, they encode them into the state transition systems $\Sigma_{W_1}, \Sigma_{W_2}, \ldots, \Sigma_{W_n}$. Then, they generate a single transition system $\Sigma$ which results from the parallel composition of $\Sigma_{W_1}, \Sigma_{W_2}, \ldots, \Sigma_{W_n}$. $\Sigma$ hence represents the possible behaviour of the given services, hence, the possible evolutions of a planning domain, which a planner suitably analyses in order to find a plan $\pi$ which satisfies the requirements expressed by a given complex goal. Finally, the plan $\pi$ (which is an automaton) is automatically translated into an executable business process.

Traverso et al. also proposed in [72] an approach to automatise the composition of (abstract) BPEL processes. Each BPEL process is enhanced with semantic annotations which link the procedural description of the process with the (separate) ontological description of the data exchanged by the process. The composition problem is hence defined as follows. First, given a set of annotated BPEL processes $P_1, P_2, \ldots, P_n$ modelled as annotated state transition systems $\Sigma_{P_1}, \Sigma_{P_2}, \ldots, \Sigma_{P_n}$, the parallel composition $\Sigma_{\parallel}$ of $\Sigma_{P_1}, \Sigma_{P_2}, \ldots, \Sigma_{P_n}$ is performed. Then, given a global ontology defining the relevant concepts of the composition scenario, the global state transition system $\Sigma_{\parallel}$, and a composition requirements specifying a “primary condition” (to be satisfied whenever possible) and a “recovery condition” (to be satisfied when the primary condition fails), the composition problem consists of generating a new transition system $\Sigma_c$ that if connected to $\Sigma_{\parallel}$ satisfies the composition requirements.

Another approach to the automatic composition of services was presented by Berardi et al. in [11]. They assume that services in a community export their behaviour in terms of a common set of actions shared by the community. The behaviour of each service is modelled as a deterministic finite transition system, that employs the actions of the community. The client request is expressed as a deterministic finite transition system as well, the so-called target service. The problem of service composition is then reduced into a problem of satisfiability of a formula of Deterministic Propositional Dynamic Logic (DPDL). In particular, a suitable DPDL formula
encoding the target services, the services in the community and some domain independent condition is constructed. Next, the satisfiability of such a formula is checked by exploiting well-known tableau algorithms. If the formula is satisfiable, a transition system modelling the composition is returned. Similarly to [81], the abstract specification of the composition is translated into executable business processes. In [12], the approach in [11] is extended to cope with services only partially controllable (hence modelled as non-deterministic transition systems), yet, fully observable by a service orchestrator.

2.2.4 Service replaceability

In this Subsection we briefly discuss some approaches which can be suitably employed to tackle the problem of service replaceability. Indeed, such approaches (implicitly) provide a replaceability relation of services. Hence, if a service \( S \), taking part in a composition \( C[S] \) can be replaced by a service \( T \), the composition \( C[T] \) works properly, as well.

A logic-based approach for service replaceability has been recently presented in [69], where a context-specific definition of service equivalence is introduced. According to [69], given a \( \mu \)-calculus formula \( \phi \) describing some property, a service \( S \), taking part in a specific context \( C[\cdot] \), can be replaced by a service \( T \) if \( \phi \) holds also in \( C[T] \). Intuitively, such a notion of context-specific replaceability relaxes the requirements imposed by a notion of service (bi)simulation like, e.g., [16].

Another relaxed replaceability relation on services is induced by the definition of interaction soundness presented in [74]. Given an environment \( E \), a service \( S \) in an orchestration \( O[S] \) can be replaced by \( T \), if the interaction of \( O[T] \) and \( E \) is lazy sound, that is, if the final node of the graph which represents the interaction of \( O[T] \) and \( E \) can be reached from every initial node.

Although not presented in term of replaceability, the notion of operating guidelines introduced in [50, 46], and employed in [45] to formally analyze the interactional behaviour of BPEL processes, also implicitly induces a replaceability relation on services — yet not compositional. An operating guideline is an automaton that concisely represents all the partners that properly interact with a service. A service \( S \) interacting with \( C \) can be replaced with a service \( T \), if \( T \) belongs to the operating guidelines of \( C \).

A theory for checking the compatibility of service contracts based on a \( \text{ccs} \)-like calculus is presented in [28, 42]. Using a simple finite syntax (featuring the sequencing and external/internal choice constructors) to describe service contracts, they define a notion of preorder on processes (based on must testing) reflecting the ability of successfully interacting with clients. Such a notion induces a replaceability relation that, informally, allows one to replace a service \( S_1 \) with \( S_2 \) only if all clients compliant with \( S_1 \) are also compliant with \( S_2 \).
2.2.5 Our previous results

Our first approach to a composition-oriented service discovery was introduced in [26] and next extended in [27]. Such a discovery methodology takes as input a registry of OWL-S described services and a client query specifying the ontology-annotated inputs and outputs of the service to be found, and it returns as output the possible sequences of atomic process invocations that the client must perform in order to achieve the desired result. The methodology consists of three main phases:

(1) *Translation of each (registry-published) service into a tree structure* — Each service is represented as a tree, where leaf and internal nodes correspond to atomic and composite processes, respectively. In particular, such trees contain only sequence, choice, split+join and split internal nodes, since the composite processes if-then-else, any-order, repeat-until, repeat-while are suitably re-defined in terms of choice and sequence control constructs in order to simplify the construction of the dependency graph (2).

(2) *Construction of a hypergraph representing the dependencies among atomic processes of the matched services* — The dependency hypergraph, which is the result of the matching phase, contains two types of nodes: process nodes and data nodes, respectively corresponding to the matched atomic processes and their inputs/outputs data. An hyperedge can connect a set $I$ of data nodes with a process node $p$ (viz., $D$ is the set of the inputs of $p$), a process node $p$ with a set $O$ of data nodes (viz., $O$ is the set of the outputs of $p$), a set $D$ of data nodes with a data node $d$ (viz., $D$ is the set of the sub-types of $d$), a set $P$ of process nodes and a process node $p$ (viz., $P$ is the set of predecessors of $p$), and a process node $p$ with a set $P$ of process nodes (viz., $P$ is the set of the processes which are in a mutual exclusion relationship with $p$). In particular, the hyperedges connecting a set $P$ of process nodes with a process node $p$ are the so-called **sequencing constraints** (viz., $p$ can be executed when each $q \in P$ has completed), while the hyperedges connecting a process node $p$ with a set $P$ of process nodes are the so-called **excluding constraints** (viz., none of the processes in $P$ can be performed if $p$ has been previously executed).

(3) *Analysis of the dependency hypergraph to determine a service composition capable to satisfy the query* — This phase consists of five steps:

(3.1) **Reachability of query outputs** — This step checks whether there are query output nodes in the hypergraph that do not have incoming hyperedges from process nodes. If such disconnected query outputs really exist, the discovery algorithm fails (since there are no services available capable of producing such query outputs).

(3.2) **Yellow colouring** — This step identifies, by colouring them in yellow, all process and data nodes which may be useful for generating the query
outputs. In particular, it first colours in yellow all the query output nodes, and then, it recursively paints in yellow all process and data nodes that are white and that belong to the tail of an hyperedge whose head includes at least one yellow coloured node.

(3.3) Red&Black colouring — This step identifies, by painting them in red, the processes which contribute to generate the query outputs and which can actually be executed if the query inputs are provided. In particular, this step firstly paints in red all the query input nodes. Then, a process node \( p \) can be painted in red, and we say that \( p \) is firable, if \( p \) is yellow and all its input data nodes are red, and if there is at least one set of predecessors of \( p \) then at least one set of predecessors of \( p \) is red coloured. When a process \( p \) is painted in red, \( p \) is added to an initially empty process sequence list, the output data nodes of \( p \) are painted in red, as well, while all the process nodes linked to \( p \) by excluding constraints are inhibited by painting them in black. Note that if there are \( n \) firable process nodes linked through excluding constraints, then the execution of this step splits in \( n \) instances, each of them painting in red one of the firable process nodes. Each instance of this step finishes either with a success if all query outputs become red, or with a failure if there are no more firable processes but there is still at least one yellow query output. As a result of this step we shall obtain a set of triples \( \langle \text{answer}, H, APL \rangle \), where \text{answer} ranges over \{SUCCESS, FAILURE\}, \( H \) is the coloured hypergraph, and \( APL \) denotes the list of atomic process invocations to be performed in order to satisfy the client query.

(3.4) Analysis of triples — If there exists at least a triple \( \langle \text{success}, H, APL \rangle \), this steps returns an ordered list of all the successful triples. Instead, if all the triples generated by step (3.3) are failures, this step checks whether there exists a set of failures that together are able to generate all outputs requested by the query. In this case we have a success obtained from the aggregation of a set of failures, otherwise, the discovery algorithm fails.

(3.5) Individuating additional inputs — If step (3.4) fails, this last step looks for additional inputs that have to be provided in order to have further firable processes that help generating the unsatisfiable query outputs.

Hence, the discovery methodology we introduced in [27] extends the approaches such as, e.g., [66, 7], which are oriented to a single service discovery, thus not searching for compositions of services. Comparing [27] with some composition-oriented approaches, such as [5] and [8], we observed that [27] – conversely to [5, 8] – analyses the process model of services (viz., service behaviour) in order to perform a finer-grained matchmaking, at the level of atomic processes inside services rather than at the level of entire services. Moreover, when no service composition can satisfy the query, [27] is also capable of suggesting additional inputs that would permit
to get a full match. Furthermore, we noticed that, while approaches such as, e.g., [57] and [36] address service composition by focussing on analysing input/output dependencies among services, [27] also considers the ordering of atomic processes (inside services) which is crucial in order to determine the behaviour of a service composition, e.g., to determine whether it may deadlock or not.

Yet, [27] presents some known limitations that we briefly discuss hereafter. First [27] does not cope with separate ontologies, since it (unrealistically) assumes that a single shared ontology is employed to semantically annotate service descriptions. It is also worth noting that [27] may return non-minimal sequences of atomic process invocations, i.e., sequences including processes not strictly necessary to satisfy the client request. Moreover, [27] does not guarantee the correct termination of the services whose atomic processes have to be invoked to satisfy the client request. Another known limitation of [27] is that it constructs the dependency hypergraph during the matching phase, thus affecting the query answering time. Furthermore, [27] is not able to process behavioural client queries, that is, queries also specifying the expected behaviour of the service to be found. Finally, it is also worth observing that the discovery algorithm introduced in [27] does not cope with scenarios which involve multiple instances of the same service. For example, consider two atomic processes AP1 and AP2 linked through an excluding constraint. AP1 generates as output $x$, while AP2 takes as input $x$ and produces $y$. A client request asking for $y$ may hence be solved by executing first AP1 and then AP2. Yet, during the red colouring phase (step 3.3) the algorithm can only paint in red AP1, and this leads to painting in black (viz., burning) AP2. As a result such an algorithm returns a failure, as it can not paint in red the output $y$ of AP2 requested by the query.

Some limitations of [27] have been addressed in [21], where we presented an approach to a behaviour-aware service discovery. Similarly to [27], the matchmaking system introduced in [21] takes as input a registry of OWL-S described services and a client query specifying the ontology-annotated inputs and outputs of the service to be found. Then, it returns as output the (Petri net specification of the) compositions of services capable of satisfying the request. In particular, such a matchmaking system consists of three independent components: a translator from OWL-S process models to (place/transition) Petri nets, a functional analyser which filters services taking into account their functional attributes, and a behavioural analyser which after merging together a set of (selected) Petri nets, checks for a positive match by animating the composite Petri net.

In general terms, the system behaviour of [21] is the following. Whenever a service is added to the registry, the translator generates (and adds to the registry) the Petri net representation of the OWL-S process model of the service, while (a module of) the functional analyser performs a registry inspection in order to synthesise the information about the outputs produced by the available services. Such information is stored in a data structure, called outputRegistry, which associates each output $o$ with the set of services outputRegistry[$o$] that generate it. Then,
CHAPTER 2. BACKGROUND AND STATE OF THE ART

When the matchmaking system receives a client request, the functional analyser exploits the information provided by the outputRegistry in order to determine all the minimal sets of services that may be useful to satisfy the query. Next, the behavioural analyser analyses (the Petri net representation of) such sets of services until it finds one (if any) capable to fulfill the request. In order to establish whether a service set can really satisfy the query, the behavioural analyser firstly performs the parallel composition of the Petri nets modelling the services contained in the candidate set. Next, it determines whether such services can be composed together, so to generate the query outputs without dead-locking, by analysing the control flow of the global Petri net. More precisely, the behavioural analyser checks the so-called may-termination of the resulting net. Indeed, it checks whether there exists (at least) a Petri net animation which terminates by generating the requested outputs. If so, the behavioural analyser returns a positive match to the client, otherwise it analyses the next candidate service set.

As one may note, [21] advances [27], since [21] guarantees the minimality and the may-termination of the successful compositions of services returned to the client. It is also worth observing that the Petri net representation of services as well as the construction of the outputRegistry is performed whenever a new service enters the local registry, thus not affecting the query answering time.

It is worth noting that the discovery technique in [21] separates the analysis of the functional information from the analysis of behavioural information and it introduces Petri nets to model service behaviour, as in the approach described in this thesis. Yet, [21] employs (place/transition) Petri nets only as an instrument to simulate/animate service behaviour, thus defining neither a formal encoding from (OWL-S) descriptions to Petri nets, nor a well-founded notion of behavioural congruence for Petri nets (viz., services). As a consequence, [21] does not cope with behavioural client requests and it can be employed neither to generate a requested service composition nor to find a (composition of) service(s) capable of safely replacing a given service. Moreover, [21] does not provide a clear methodology to verify the may-termination (the must-termination is not considered) of the selected services, instead, it assumes the possibility to check in some way the existence of some terminating Petri net animation. Finally, it is worth noting that the outputRegistry employed to classify services with respect to the produced outputs, may contain redundant information. Indeed, consider two concepts $c$ and $c'$, with $c'$ sub-concept of $c$. Then, outputRegistry[$c'$] contains all the services which generate $c'$, yet, outputRegistry[$c$] contains all the services which generate $c$ and the services producing $c'$ (viz., outputRegistry[$c'$]). Moreover, [21], as [27], does not cope with the “crossing” of separate ontologies, hence assuming a single global, shared ontology. As we will discuss in Chapters 3 and 4, the approach described in this thesis does not suffer from all such limitations.
2.2.6 Discussion

We conclude this Section with a comparative synthesis of the state-of-the-art approaches described so far. Note that a more focussed discussion of the (comparative) advantages of the approach described in this thesis is delayed until Sections 3.3 and 4.5, when the key components of our discovery technique will have been introduced. In Table 2.1, we classified the literature approaches with respect to the issues that they address (viz., single service discovery, composition-oriented service discovery, service composition and service replaceability) and, orthogonally, with respect to the information that they use (viz., semantics, behaviour, semantics and behaviour). Moreover, we also considered (where possible) the type of the queries that they address. In particular, in Table 2.1, “F” denotes functional queries (viz., query specifying inputs and outputs of the desired service), while “B” denotes behavioural queries (viz., query specifying inputs, outputs and expected behaviour of the desired service).

<table>
<thead>
<tr>
<th>Approach</th>
<th>Query</th>
<th>Semantics</th>
<th>Behaviour</th>
<th>Semantics and behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single service discovery</td>
<td>F</td>
<td>[44, 66, 79, 41]</td>
<td></td>
<td>[7]</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td>[65, 1]</td>
</tr>
<tr>
<td>Composition-oriented</td>
<td>F</td>
<td>[5, 8]</td>
<td></td>
<td>[36, 61, 27, 21]</td>
</tr>
<tr>
<td>service discovery</td>
<td></td>
<td></td>
<td></td>
<td>[57]</td>
</tr>
<tr>
<td>Service composition</td>
<td></td>
<td></td>
<td></td>
<td>[34, 11, 12]</td>
</tr>
<tr>
<td>Service replaceability</td>
<td></td>
<td>[69, 45, 74, 28, 42]</td>
<td>[81, 72]</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Classification of the state-of-the-art approaches.

As anticipated in Subsection 1.2, the primary objective of this thesis is to define a suitable technique for a behaviour- and ontology-aware service discovery, capable of generating service compositions and of finding (compositions of) services behaviourally equivalent to a desired service.

In this setting, a key requirement is hence the ability of addressing behavioural queries. As one may note in Table 2.1, only the approaches in [65, 1, 57] cope with behavioural queries, which are expressed by means of temporal logic, \( \pi \)-calculus and finite state automata, respectively. Yet, [65, 1] focus on a single service discovery, while, as we discussed in the Introduction, the satisfaction of complex queries may need to compose the functionalities of several services. A composition-oriented discovery is the goal of [57], that, yet, considers a very strict match between the query automaton and the composition automaton, thus not allowing to match a service operation (e.g., taking as input \( A \) and producing \( B \)) with a suitable interaction of several operations (e.g., a sequential composition of two operations, the first taking \( A \) and producing \( C \), and the second taking \( C \) and producing \( B \)). Most state-of-the-art approaches to composition-oriented discovery address functional client queries only. In particular, [5, 8] only consider the possibility of using several (non inter-
acting) services to produce the requested outputs, [36] selects (possibly interacting services) not violating dependencies among their inputs and outputs, [61] considers only services which can be invoked sequentially, while [27] features more flexible service compositions. Yet, none of the approaches in [5, 8, 36, 61, 57, 27] guarantees the termination of the selected (and possibly composed together) services. Only the approach in [21] checks whether the selected (composition of) service(s) may possibly terminate.

As one can note in Table 2.1, all the (cited) approaches addressing (single/composi-
tion-oriented) service discovery exploit semantics information, since it is necessary to feature high-level, flexible matches. Yet, we stress that semantics information in isolation does not suffice to satisfy complex requests as well as to ensure the generation of not locking service compositions.

Conversely, the approaches addressing service composition, such as [34, 11, 12], focus on service behaviour to properly compose a given set of services. Semantics information is however exploited in [81, 72], in order to automatically determine correspondences between inputs and outputs of the services to be composed. The approaches in [34, 11, 12, 81, 72] provide suitable techniques to compose a pre-selected set of services so as to achieve the required composition, yet, they do not consider how such a set can be suitably discovered. This is the case also of the approaches to service replaceability, such as [69, 45, 74, 28, 42] which focus on proposing suitable techniques to check whether a service can be replaced by another service, not taking into account how to find candidate (compositions of) services which can potentially replace the original service. Moreover, [69, 45, 74, 28, 42] do not consider possibly syntactic mismatches between a candidate service and the service to be replaced, since they focus only on service behaviour.

In the following Chapters 3 and 4, we present a composition-oriented discovery technique that first exploits semantics information to select (sets of) services capable of satisfying the functional requirements of the client request, and next, it exploits behaviour information to generate a composition of the selected services which is guaranteed to terminate and to be equivalent to the behaviour (possibly) specified in the client request. With respect to Table 2.1, the work presented in this thesis surely belongs to the composition-oriented discovery approaches, which exploit semantic and behaviour information to satisfy (functional and) behavioural requests. Note, however, that our work provides some partial contributions to service composition and to service replacement, as well.
Chapter 3

Functional analysis

The service discovery technique presented in this thesis exploits both functional and behavioural information published in (OWL-S) service descriptions. With the term “functional analysis” we denote the parts of the discovery technique that cope with functional information. In particular, the functional analysis, described in this Chapter, includes:

1. the synthesis in a suitable data structure of functional dependencies within and among services, and

2. the selection of services with respect to their functional attributes.

Briefly, step (1) processes the description of a service when a provider registers such a service to the local registry. Step (1) determines the functional dependencies within the new service as well as the functional dependencies among the new service and the previously registered services, and it collects such dependencies into a suitable data structure. In particular, step (1) builds a dependency hypergraph, whose nodes represent functional attributes of services, and whose hyperedges represent relationships among them. Step (2) takes as input a client query and it explores the dependency hypergraph in order to select (sets of) services which satisfy the functional requirements of the query.

The Chapter is organised as follows. Step (1) is described in Section 3.1. We introduce the dependency hypergraph in Subsection 3.1.1, and we discuss how to construct such a hypergraph in Subsection 3.1.2. A sample dependency hypergraph is introduced in Subsection 3.1.3. Next, Section 3.2 is devoted to the description of step (2). We describe the algorithm for discovering (sets of) services in Subsection 3.2.1, and we discuss the ability of such an algorithm of discovering minimal set of services in Subsection 3.2.2. Soundness, completeness and complexity of the algorithm are discussed in Subsection 3.2.3. Then, we introduce in Subsection 3.2.4 two sample client queries that we will use to illustrate our approach throughout this thesis. In particular, Subsection 3.2.4 describes how the algorithm satisfies the functional requirements of the sample queries. Finally, we present some concluding remarks in Section 3.3.
Preliminary versions of the results presented in this Chapter have been published in [24, 23, 22].

### 3.1 Collecting functional information

As anticipated in the Introduction, with *functional information* of services we mean the information regarding the functional attributes of services, namely, their input and output parameters. In particular, functional information defines functional *dependencies* within and among services. Hence, we need to represent the relationships that state “which set of inputs a service requires in order to produce a set of outputs” (viz., *intra*-service dependencies), as well as “which set of outputs produced by a service is required as input by another service” (viz., *inter*-service dependencies).

Furthermore, in our context, where we consider *semantic* Web services advertised by means of OWL-S descriptions, each input and output parameter is annotated with a concept defined in some available ontology. Hence, it is also necessary to represent the *semantic* relationships among concepts, such as, for instance, those relationships that state “which are the sub-concepts (or equivalent concepts) of a given concept”.

We collect all the above mentioned types of functional dependencies into a *hypergraph*, which is a data structure suitable to easily represent such dependencies by means of nodes (viz., functional attributes of services) and hyperedges (viz., relationships among functional attributes). Before describing the *dependency hypergraph* (Subsection 3.1.1) and how to construct it (Subsection 3.1.2), we include hereafter the definitions of hypergraph, directed hypergraph and directed hyperedge (Figure 3.1), as formally described in [33].

**Definition 3.1 (hypergraph).** A hypergraph is a pair $H = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of vertices (or nodes) and $E = \{E_1, E_2, \ldots, E_m\}$, with $E_i \subseteq V$ for $i = 1, \ldots, m$, is a set of hyperedges. Note that when $|E_i| = 2$, $i = 1, \ldots, m$, the hypergraph is a standard graph.

**Definition 3.2 (directed hypergraph).** A directed hypergraph is a hypergraph with directed hyperedges. A directed hyperedge is an ordered pair, $E = (X, Y)$, of (possibly empty) subsets\(^1\) of vertices. $X$ is the tail of $E$, denoted by $T(E)$, while $Y$ is its head, denoted by $H(E)$.

---

\(^1\)While in [33] $X$ and $Y$ are required to be *disjoint* sets, we observe that such a constraint is not needed in our context. Intuitively, a directed hyperedge ($\{c\}, \{c\}$) denotes a service which inputs (a parameter of type) $c$ and generates (a parameter of type) $c$. 
3.1. COLLECTING FUNCTIONAL INFORMATION

3.1.1 The dependency hyperedge

The dependency hypergraph is a labelled directed hypergraph which collects the functional dependencies within/among the services stored in a local registry. A labelled directed hypergraph is a hypergraph with labelled hyperedges.

**Definition 3.3 (labelled hypergraph).** A labelled directed hypergraph \((E, V, l)\) is a directed hypergraph \((E, V)\) with a labeling function \(l : E \rightarrow A\) assigning to each hyperedge a label from a given alphabet \(A\).

A labelled directed hyperedge is denoted by a triple \(E = (X, Y, a)\), where \(X, Y, a\) denote the tail, the head and the label of \(E\), respectively.

The vertices of the dependency hypergraph correspond to the concepts defined in the ontologies employed by the service descriptions, while the hyperedges represent relationships among such concepts. In particular, an hyperedge has one of the following three types:

- \(E_C = (D, \{c\}, \text{nil})\) – subConcept relationship. Let \(c\) be a concept defined in an ontology \(O\) and let \(D \in O\) be the set of the (direct) sub-concepts of \(c\). Then, there is a \(E_C\) hyperedge from \(D\) to \(c\).

- \(E_S = (\{e\}, \{f\}, \text{sim})\) – equivalentConcept relationship. Let \(e, f\) be two concepts defined in two separate ontologies and let \(\text{sim}\) be the similarity between \(e\) and \(f\), i.e., the probability that \(e\) is (semantically) equivalent to \(f\). If \(\text{sim}\) is above a given similarity threshold, there is a \(E_S\) hyperedge from \(e\) to \(f\) labelled by \(\text{sim}\).

- \(E_S = (I, O, s)\) – intra-service dependency. Let \(s\) be (a profile of) a service and let \(I\) be the set of the inputs that \(s\) requires to produce the set \(O\) of the outputs. Then, there is a \(E_S\) hyperedge from \(I\) to \(O\) labelled by \(s\).

We now discuss how to determine semantic relationships among concepts defined in different ontologies as well as how to compute the intra-service dependencies of a given service.
Determining *subConcept* and *equivalentConcept* relationships

The dependency hypergraph represents two types of semantic relationships between ontology concepts, namely, *subConcept* and *equivalentConcept* relationships.

*SubConcept* relationships can be determined within an ontology. More precisely, given two concepts \( c \) and \( d \) defined in the same ontology, we say that \( c \) is a *subConcept* of \( d \), if \( c \) is more specific than \( d \). For example, "car" is more specific than "vehicle", so that "car" is a *subConcept* of "vehicle". With respect to the hierarchical structure of an ontology, \( c \) is a *subConcept* of \( d \) if \( c \) is a child of \( d \).

*EquivalentConcept* relationships can be established among concepts defined in separate ontologies. Given two concepts \( c_1 \) and \( c_2 \), defined in the ontologies \( O_1 \) and \( O_2 \), respectively, we say that \( c_1 \) is *equivalent* to \( c_2 \) if \( c_1 \) is semantically equivalent to \( c_2 \), that is, if \( c_1 \) and \( c_2 \) feature the same semantic meaning. For example, "price" can be considered equivalent to "charge" and to "cost", since they are synonyms.

To determine *subConcept* and *equivalentConcept* relationships, we exploit the ability of reasoning with (separate) ontologies of the SemFiT [39] Web service. SemFiT is able to handle ontology repositories, and in particular, it provides suitable operations for finding (direct) sub-concepts and equivalent concepts of a given ontology concept. We briefly describe the main SemFiT operations in Subsection 3.1.2.

It is worth discussing here how SemFiT establishes equivalence relationships between concepts. SemFiT combines the results of different individual matchers which analyse the similarity between pairs of concepts with different strategies:

1. **name similarity** – compares the names of the provided concepts,

2. **data type similarity** – compares the types of the provided concepts, and establishes a similarity value depending on the possibility of producing each type from another type by casting,

3. **WordNet similarity** – checks whether the two concepts are synonyms in WordNet [75],

4. **path similarity** – compares the path from each concept to its root in the ontology, and provides an average between the path length similarity and the name similarity between all concepts in both paths,

5. **property similarity** – compares the similarities among the properties of the concepts compared, and

6. **edit distance similarities** – it is based on the comparison of strings (i.e., the names of the provided concepts), and calculates the difference between them as the operations that should be performed on one string to obtain another.

It is also interesting to note that, by taking into account the similarities between their concepts, SemFiT calculates the distance between pairs of ontologies. A group of interrelated ontologies, that is, a group of ontologies whose distance values exceed
3.1. COLLECTING FUNCTIONAL INFORMATION

a previously established threshold, constitutes a semantic field [60]. Let us consider a concept \( c \) defined in an ontology \( O \). When \text{SemFiT} is asked for finding the equivalent concepts of \( c \), it searches for equivalent concepts only within the ontologies belonging to the semantic field of \( O \).

The extended notion of subConcept relationship

The subConcept relationship considered so far holds between two concepts defined in the same ontology. We now extend the notion of subConcept over concepts defined in (possibly) different ontologies.

In the following definition, \( \sqsubseteq \) denotes the reflexive subConcept relation defined within an ontology, while \( \equiv \) denotes the equivalence relation among concepts belonging to possibly different ontologies.

**Definition 3.4 (subConcept).** A concept \( c \) is a subConcept of a concept \( d \), and we write \( c \sqsubseteq d \) if and only if

1. \( c \sqsubseteq d \), or
2. \( c \equiv c' \) and \( c' \sqsubseteq d \), or
3. \( c \sqsubseteq d' \) and \( d' \equiv d \).

Hence, \( c \) is subConcept of \( d \) if \( c \) is a subConcept of \( d \) in some ontology (1), or if \( c \) is equivalent to \( c' \) and \( c' \) is a subConcept of \( d \) in some ontology (2), or if \( c \) is a subConcept of a \( d' \) in some ontology and \( d' \) is equivalent to \( d \) (3).

We now reformulate Definition 3.4 in terms of the dependency hypergraph. Namely, \( c \) is subConcept of \( d \) (i.e., \( c \sqsubseteq d \)) if and only if there exists a path from \( c \) to \( d \) which consists of subConcept relationships (viz., \( E_{\sqsubseteq} \) hyperedges) and/or equivalentConcept relationships (viz., \( E_{\equiv} \) hyperedges).

**Definition 3.5 (subConcept).** Given a hypergraph \( H = (V, E) \), a concept node \( c \) is a subConcept of a concept node \( d \), and we write \( c \sqsubseteq d \) if and only if

\[
\exists e_1, e_2, ..., e_n \in (E_{\equiv} \cup E_{\sqsubseteq}) : c \in T(e_1) \wedge \{d\} = H(e_n) \wedge H(e_i) \subseteq T(e_{i+1}) \forall i \in [1, n - 1].
\]

Computing the intra-service dependencies

A Web service normally provides different operations, which are composed together by means of few control constructs, such as, for instance, sequence, choice, if-then-else, iterate and so on. The complete behaviour of a service may hence be arbitrarily complex. Let us now consider a (very) simple service \( c \), which takes as input \( in \) and returns as output \( out_1 \) or \( out_2 \). As one may note, \( c \) implements a choice and it can behave in two different ways, that is, as a service \( c_1 \) which inputs \( in \) and produces \( out_1 \), and as a service \( c_2 \) which inputs \( a \) and produces \( out_2 \).
We can hence observe that a service may behave in different ways and feature different functionalities. To characterise a particular behaviour \( b \) of a service \( s \) from a functional point of view, we can represent it as a triple \((I_b,O_b,b)\), where \( I_b \), \( O_b \) respectively denote the set of the inputs and the set of the outputs employed by the service \( s \) when it features the behaviour \( b \). For instance, consider again the previously introduced choice service \( c \). The two behaviour of \( c \), viz., \( c_1 \) and \( c_2 \), can be represented by the triples \((\{\text{in}\},\{\text{out}_1\},c_1)\) and \((\{\text{in}\},\{\text{out}_2\},c_2)\), respectively.

An intra-service dependency is a triple \((I_b,O_b,b)\) which identifies a particular behaviour \( b \) of a service \( s \). In other words, \( I_b \) is the set of the inputs required by \( b \) in order to produce the set \( O_b \) of the outputs. In the dependency hypergraph, an intra-service dependency \((I_b,O_b,b)\) is a hyperedge, labelled by \( b \), which connects two sets of nodes, respectively corresponding to the set \( I_b \) of the inputs and to the set \( O_b \) of the outputs of \( b \).

In the rest of this thesis, we shall use the term “profile” to name “intra-service dependency”. Note that we intentionally use the term “profile”, since intra-service dependencies are reminiscent of the OWL-S profiles (Section 2.1.1). It is important to observe, however, that while an OWL-S profile is a synthesis of a service (usually excluding those inputs and outputs not relevant to explain the service functionality), an intra-service dependency collects all the inputs and all the outputs of a (computation of a) service.

In Figure 3.2 we show how it is possible to determine the profiles of a service by suitably analysing its OWL-S process model, that, as described in Section 2.1.1, details the service behaviour. Figure 3.2 presents the recursive MakeProfiles function, which takes as input the OWL-S process model of a service, and determines its intra-service dependencies. Let us consider the tree-view of the OWL-S process model, that we introduced for the sample services presented in Section 2.1.1 (e.g., see Figure 2.3). MakeProfiles is initially invoked over the root composite process of a service, and next, it recursively analyses each child composite process, until reaching the leaf processes (viz., the atomic processes). MakeProfiles returns a set of pairs \(\{(I_1,O_1),(I_2,O_2),..., (I_n,O_n)\}\), each pair \((I_b,O_b)\) (with \(1 \leq b \leq n\)) representing the profile (i.e., the set \( I_b \) of the inputs and the set \( O_b \) of the outputs) of a behaviour \( b \) of a given service.

Straightforwardly, when MakeProfiles executes on an atomic process \( A \), it returns a single pair \((I,O)\) where \( I \) and \( O \) denote the set of the inputs (viz., \( \ln(A) \)) and the set of the outputs (viz., \( \out(A) \)) of the process, respectively. Instead, when MakeProfiles executes on a process if-then-else \((P_1,P_2)\), it returns the pairs \(\{(I,O) | (I,O) \in \MakeProfile(P_1)\}\) and the pairs \(\{(I,O) | (I,O) \in \MakeProfile(P_2)\}\) which functionally characterise the left child process \( P_1 \) and the right child process \( P_2 \), respectively. Similarly, when MakeProfiles executes on a process choice \((P_1,P_2,...,P_k)\), it returns \( k \) sets of pairs \(\{(I,O) | (I,O) \in \MakeProfile(P_i)\}\) with \(1 \leq i \leq k\), each of them featuring the profiles of the child process \( P_i \).

Let us now consider a process sequence \((P_1,P_2,...,P_k)\). According to the OWL-S process model, all the child processes \( P_1,P_2,...,P_k \) of the sequence process have
3.1. COLLECTING FUNCTIONAL INFORMATION

- MakeProfiles(atomic(A)) = { (In(A), Output(A)) }
- MakeProfiles(if-Then-Else(\(P_1, P_2\))) = \(\bigcup_{i=1}^{2} \{ (I, O) \mid (I, O) \in \text{MakeProfiles}(P_i) \} \)
- MakeProfiles(choice(\(P_1, ..., P_k\))) = \(\bigcup_{i=1}^{k} \{ (I, O) \mid (I, O) \in \text{MakeProfiles}(P_i) \} \)
- MakeProfiles(sequence(\(P_1, ..., P_k\))) = MakeProfiles(split(\(P_1, ..., P_k\)))
  = MakeProfiles(split+join(\(P_1, ..., P_k\))) = MakeProfiles(any-order(\(P_1, ..., P_k\)))
  = \{ (I, O) \mid \exists (I_1, O_1), (I_2, O_2), ..., (I_k, O_k) : (I_i, O_i) \in \text{MakeProfiles}(P_i) \ \forall i \in [1, k] \}
  \land (I = I_1 \cup I_2 \cup ... I_k) \land (O = O_1 \cup O_2 \cup ... O_k) \}
- MakeProfiles(repeat-until(\(P\)))
  = MakeProfiles(choice(\(P\), sequence(\(P, P\), ..., sequence(\(P, ..., P\))))\text{ Alt}(\(P\))
- MakeProfiles(repeat-while(\(P\))) = \{ (\{\}, \{\}) \}
  \cup MakeProfiles(choice(\(P\), sequence(\(P, P\), ..., sequence(\(P, ..., P\))))\text{ Alt}(\(P\))

Figure 3.2: Pseudo-code for computing the profiles of a service.

...to be necessarily executed. Hence, if \((I, O)\) is a profile of sequence(\(P_1, P_2, ..., P_k\)), then \(I\) and \(O\) result from the union of the sets \(I_1, I_2, ..., I_k\) and \(O_1, O_2, ..., O_k\), respectively, where each \(I_i\) and \(O_i\), with \(1 \leq i \leq k\), denotes one of the possible sets of inputs and one of the possible set of outputs of the child process \(P_i\). In particular, let \(x_i\) denote the number of the pairs \((I, O)\) computed by MakeProfiles for the child process \(P_i\), with \(1 \leq i \leq k\). Then, when MakeProfiles executes on a process sequence(\(P_1, P_2, ..., P_k\)), it returns \(n = \prod_{i=1}^{k} x_i\) pairs \((I_j, O_j)\) with \(1 \leq j \leq n\). Similarly to the sequence control construct, processes split, split+join and any-order complete only after executing all their child processes. Hence, MakeProfiles(split(\(P_1, P_2, ..., P_k\))), MakeProfiles(split+join(\(P_1, P_2, ..., P_k\))) as well as MakeProfiles(any-order(\(P_1, P_2, ..., P_k\))) feature the same behaviour of MakeProfiles when executed on a sequence(\(P_1, P_2, ..., P_k\)) process.

Before describing how MakeProfiles works with repeat-until processes, it is important to note that this functional characterisation of services focuses on which inputs and outputs are required and produced by the services, while it abstracts from how many instances of such inputs and outputs are really required and produced. We can hence expand a process repeat-until(\(P\)) into a choice of sequences, where each sequence is a possible alternative execution of \(P\), as illustrated in Figure 3.2. Hence, executing MakeProfiles over a repeat-until process translates into executing MakeProfiles over a choice process. Similarly, we expand a repeat-while process into a choice of sequences, as well. Yet, the choice among all possible alternative
executions of $P$ also includes a *nop* branch, since $P$ may be skipped altogether.

We have now to determine – given a process $P$ – how many alternative executions of $P$ are needed in order to execute, at least once, all the child atomic processes of $P$ (so to generate all the outputs producible by $P$). Let $\text{Alt}(P)$ be the number of the necessary alternative executions of $P$. This means that, after $\text{Alt}(P)$ alternative executions of $P$, all the child atomic processes of $P$ have been executed, so that we can properly compute the set of profiles of the process \texttt{repeat-until}(P).

\begin{itemize}
  \item $\text{Alt}($atomic$(A)) = 1$
  \item $\text{Alt}($ifThenElse$(P_1, P_2)) = \text{Alt}(P_1) + \text{Alt}(P_2)$
  \item $\text{Alt}($choice$(P_1, \ldots, P_n)) = \text{Alt}(P_1) + \ldots + \text{Alt}(P_n)$
  \item $\text{Alt}($sequence$(P_1, \ldots, P_n)) = \text{Alt}($any-order$(P_1, \ldots, P_n))$
  \hfill = \text{Alt}($split$(P_1, \ldots, P_n)) = \text{Alt}($split+join$(P_1, \ldots, P_n))$
  \hfill = \max\{\text{Alt}(P_1), \ldots, \text{Alt}(P_n)\}
  \item $\text{Alt}($repeat-while$(P)) = \text{Alt}($repeat-until$(P)) = 0$
\end{itemize}

Figure 3.3: Pseudo-code for computing the alternative executions of a process.

The $\text{Alt}$ recursive function is defined in Figure 3.3. If $P$ is an atomic process, $\text{Alt}(P)$ obviously returns 1. If $P$ is a *choice* (or *if-then-else*) process, the execution of $P$ needs to be repeated $\text{Alt}(P_i)$ times, for each child process $P_i$ of $P$. If $P$ is a *sequence* (or *any-order*, *split*, *split+join*) process, $P$ needs to be repeated $P_i$ times, where $P_i$ is the child of $P$ with the highest value of $\text{Alt}(P_i)$. Finally, note that $\text{MakeProfiles}$ is a recursive function and that if a *repeat-until*/*repeat-while* process has a *repeat-until*/*repeat-while* ancestor $Q$, $\text{MakeProfiles}$ first expands $Q$ and then $P$. That is why we set $\text{Alt}($repeat-until$(P)) = 0$, and $\text{Alt}($repeat-while$(P)) = 0$ in the pseudo-code of Figure 3.3.

A detailed example illustrating the behaviour of the functions $\text{MakeProfiles}$ and $\text{Alt}$ will be described in Subsection 3.1.3.

**Functional dependencies among different services**

There is a functional dependency between two services $s$ and $t$, if $s$ produces at least a concept that is taken as input by $t$. Yet, the OWL-S specification [64] states that two processes are type compatible if for each output of one that flows to the input of the other, the type of the output is a subtype of the type of the input, that is, the output is a *sub-concept* of the input.

Then, more generally, there is a functional dependency between two services $s$ and $t$, if $s$ produces at least a concept that is a *subConcept* (according to Definition
3.1. COLLECTING FUNCTIONAL INFORMATION

3.4) of a concept taken as input by \( t \). Functional dependencies among different services are natively represented in the dependency hypergraph, as shown by the following definition.

**Definition 3.6 (dependencies among services).** Let \( s, t \) be two services, and let \( e_s, e_t \) be two profiles of \( s \) and \( t \), respectively (viz., \( e_s, e_t \in E_S \)). There is a functional dependency between \( s \) and \( t \), if \( \exists c_s, c_t : c_s \in H(e_s) \land c_t \in T(e_t) \land c_s \preceq c_t \).

3.1.2 Constructing the dependency hypergraph

Whenever a new service is added to the local registry, the dependency hypergraph has to be updated accordingly. The `AddService` function presented in Figure 3.4 summarises the update procedure of the dependency hypergraph. Generally speaking, given a new service \( s \), `AddService` firstly adds to the hypergraph all the concepts defined in the ontologies employed by the (OWL-S) description of \( s \), and it draws the hyperedges representing the `subConcept` and the `equivalentConcept` relationships. Next, `AddService` computes and adds to the hypergraph the `intra-service` dependencies of the new service.

As anticipated in Subsection 3.1.1, we invoke the methods of the `SemFiT` Web service in order to cross ontologies, that is, in order to determine the (subConcept and) equivalentConcept relationships among concepts defined in separate ontologies. Hence, before describing in detail the behaviour of the `AddService` function, we discuss how to properly interact with the `SemFiT` service.

**Interacting with the `SemFiT` Web service**

The `SemFiT` service relies on the notion of user session. `SemFiT` associates a session to each user, where he can insert new ontologies, without affecting the sessions of the other users. `SemFiT` provides the following methods:

1. `startSession()` – returns a `sessionId` identifier for the user session.
2. `insertOntology(string ontology, int sessionId)` – inserts a new ontology (if not already present) in the user session.
3. `getOntologies(int sessionId)` – returns the ontologies of the user session.
4. `getEquivalentConcepts(string ontology, string concept, int sessionId)` – returns those concepts, defined in the ontologies of the user session, which are semantically equivalent to a given concept of an ontology of the user session.
5. `getChildConcepts(string ontology, string concept, int sessionId)` – returns the (direct) sub-concepts of a given concept of an ontology of the user session.

---

We described here only the `SemFiT` methods invoked by the `AddService` function. A complete description of the `SemFiT` service can be found in [39].
(6) GETROOTCONCEPTS(string ontology, int sessionID) – returns the root concepts of a given ontology of the user session. A concept \( r \) of an ontology \( O \) is a root concept if there does not exist a concept \( c \in O \) such that \( r \) is a sub-concept of \( c \).

To properly interact with SemFiT, it is necessary to obtain a user identifier, which is required to invoke the SemFiT methods. A user identifier univocally identifies an user session. Hence, a SemFiT user has to start the interaction with SemFiT by invoking the STARTSESSION method (1), thus obtaining a sessionID (viz., a user identifier). Next, the SemFiT user can insert new ontologies to his session (2), and he can ask SemFiT for a list of the ontologies in his session (3). Then, the user can ask SemFiT for finding relationships within a given ontology of his session. For instance, he can ask for the sub-concepts of a given concept (5), and he can ask for the root concepts (6). Similarly, the user can query SemFiT to find relationships among separate ontologies of his session. In particular, given a concept, defined in some ontology of his session, the user can ask SemFiT for searching among the ontologies of his session for the concepts (semantically) equivalent to the given concept (5).

Adding a new service to the dependency hypergraph

The behaviour of the AddService function is summarised by the pseudo-code presented in Figure 3.4. Note that in Figure 3.4, we use capital letters to identify the SemFiT methods, while sessionID denotes the session identifier assigned to the functional analysis by the STARTSESSION method of SemFiT. Let us assume that — before starting the construction of the dependency hypergraph — the functional analysis initialised its session with SemFiT by suitably invoking the STARTSESSION method.

Whenever a new service is added to the local registry, the AddService function is executed. AddService takes as input three parameters (line 1), namely, the hypergraph \( H \), a service \( s \) (whose hypergraph representation has to be added to \( H \)), and the session identifier sessionID, previously assigned to AddService by the SemFiT service. AddService invokes (line 2) the supporting AddOntology function for each ontology \( O \) employed by the service \( s \). AddOntology adds to the hypergraph the concepts defined in \( O \) and it interacts with SemFiT in order to draw the hyperedges representing the subConcept and equivalentConcept relationships. Next, AddService computes the profiles of \( s \) (line 3), as described in the MakeProfiles function shown in Figure 3.2, and it inserts them in to the dependency hypergraph.

Let us now discuss in more details the AddOntology function (lines 4–20). Similarly to AddService, AddOntology takes as input three parameters (line 4), that is, the hypergraph \( H \), an ontology \( O \), and the sessionID necessary to interact with SemFiT. Firstly, AddOntology checks whether the ontology \( O \) already belongs to the session of SemFiT associated to AddService and identified by sessionID (line 5). If so, AddOntology terminates, otherwise it inserts \( O \) into the sessionID SemFiT
1. AddService(hypergraph $H = (V, E)$, service $s$, int $sessionID$)
2.   
3.     forall ontology $O$ referred by $s$ do AddOntology($H = (V, E), O, sessionID$);
4.     
5.     forall pairs ($I_n, O_n$) in MakeProfiles($s$) do Add($I_n, O_n, s_n$) to $E_S$;

4. AddOntology(hypergraph $H = (V, E)$, ontology $O$, int $sessionID$)
5.   
6.     if $O \notin$ getOntologies($sessionID$) then
7.     
8.         insertOntology($O, sessionID$);
9.     
10.    $C = \{ e : c \in$ getRootConcepts($O, sessionID$) $\}$;
11. 
12.    forall concept $c$ in $C$ do Add $c$ to $V$;
13. 
14.    while $C \neq \emptyset$ do
15.        $c = extract(C)$;
16.        forall ($e, similarity$) in getEquivalentConcepts($O, c, sessionID$) do
17.            if $e \in V$ then
18.                Add ($\{c\}, \{e\}, sim$) to $E_\equiv$;
19.            Add ($\{e\}, \{c\}, sim$) to $E_\equiv$;
20.        $D = \{ d : d \in$ getChildConcepts($O, c, sessionID$) $\}$;
21.    
22.    forall concept $d$ in $D$ do
23.        if $d \notin V$ then
24.            Add $d$ to $V$;
25.            $C = C \cup \{d\}$;
26.        if $D \neq \emptyset$ then Add ($D, \{c\}, nil$) to $E_\in$;

Figure 3.4: Pseudo-code for building the dependency hypergraph.

session (line 6). Next, AddOntology inserts the concepts defined in the ontology $O$ to the hypergraph. It starts by adding to $H$ the root concepts of $O$ (lines 7–8) and then it continues recursively, by adding the direct sub-concepts of the previously inserted concepts (lines 15–19) until reaching the leaf concepts of (the hierarchical structure of) $O$. When a new concept $c$ is added to the hypergraph, AddOntology computes the equivalentConcept relationships between $c$ and the concepts already belonging to $H$, and inserts them to $H$ (lines 11–14). Moreover, after inserting to $H$ all the sub-concepts of a concept $c$ (lines 16–18), AddOntology adds to $H$ a new subConcept relationship linking the sub-concepts of $c$ to $c$ (line 20).

Note that all the concepts in the ontologies employed by $s$ will be included in the hypergraph, independently of whether they directly occur in the specification of (some process in) $s$.

---

3If a concept $c \in O_1$ is already present in the hypergraph, a concept $c \in O_2$ is however inserted to the hypergraph, since even if $c \in O_1$ and $c \in O_2$ are syntactically (and possibly semantically) equivalent, they belong to separate ontologies and hence denote separate concepts. Note that, obviously, $c \in O_1$ and $c \in O_2$ – if semantically equivalent – will be considered as such by SemFiT.
3.1.3 An example

This Subsection presents an example which illustrates how the AddService function (Figure 3.4) builds the dependency hypergraph.

We consider an empty service registry, where we add four of the sample services introduced in Subsection 2.1.2 in the following order: ConferenceService, HotelService, RatingOne and RatingTwo. We further assume that a session of the SemFiT service has been already associated with the AddService function.

As one can observe in Figure 2.7, the ConferenceService process model employs concepts defined in two separate ontologies, namely the e-commerce and the event ontology. For each of them, AddService invokes the supporting function AddOntology. The first call of AddOntology inserts e-commerce to the SemFiT session associated to AddService, and next, it adds the concepts and the subConcept relationships defined in the e-commerce ontology to the hypergraph.

The e-commerce ontology has four root concepts, that is, charge, contactInformation, paymentMethod and document. Furthermore, charge has three sub-concepts (viz., tax, price, registrationFee) as well as paymentMethod (viz., cash, creditCard, bankAccount), document has five sub-concepts (viz., registrationReceipt, purchaseOrder, invoice, deliveryConfirmation, confirmationOrder), while contactInformation has no sub-concepts. Figure 3.5 depicts the dependency hypergraph after inserting the e-commerce ontology.

Similarly, the second call of AddOntology inserts event in to the AddService session of SemFiT, and next, it adds the concepts and the subConcept relationships defined in the event ontology to the hypergraph. The event ontology has three root concepts, namely, event, location and dateTime. event has two sub-concepts, viz., socialEvent and academicEvent. The latter, in turn, has two sub-concepts, viz., workshop and conference. Moreover, conference has two sub-concepts, as well, viz., internationalConference and europeanConference. Finally, also location and dateTime have sub-concepts, viz., city and country, and, startDate and endDate, respectively.

![Diagram of the hypergraph after inserting the e-commerce ontology.](image-url)
3.1. COLLECTING FUNCTIONAL INFORMATION

The dependency hypergraph resulting from the insertion of the event ontology is shown in Figure 3.6. Note that no semantic correspondence among the concepts of the e-commerce and the event ontologies can be established.

Next, AddService invokes the MakeProfiles function in order to compute the intra-service dependencies of the ConferenceService. As one can note in Figure 2.7, ConferenceService features two possible behaviour, which are determined by the choice composite process. Hence, ConferenceService is synthetised by two profiles, that we name $C_1$ and $C_2$, which correspond to executing the processes \{selectEvent, signin, payWithCreditCard\} and \{selectEvent, signin, bankTransfer\}, respectively.

More precisely, $C_1$ is the label of a $E_S$ hyperedge (viz., a intra-service dependency) which links the set of (input) concepts (academicEvent, contactInformation, creditCard) to the set of (output) concepts (registrationFee, city, country, startDate, endDate, registrationReceipt). Similarly, $C_2$ is the label of a $E_S$ hyperedge which links the set of (input) concepts (academicEvent, contactInformation, bankAccount) to the set of (output) concepts (registrationFee, city, country,(startDate, endDate, registrationReceipt). Figure 2.7 shows the dependency hypergraph, once the insertion of ConferenceService has been completed.

We add now the HotelService to the local service registry. As illustrated in Figure 2.6, the HotelService process model employs two different ontologies, that is, the e-commerce and the hotel ontologies. When AddService invokes AddOntology in order to insert the e-commerce ontology to the hypergraph, AddOntology directly stops, since e-commerce already belongs to the AddService session of SemFiT (hence, it has been already added to the hypergraph). Then, AddOntology is invoked again to insert the hotel ontology.

The hotel ontology has six root concepts, viz., physicalLocation, infoHotel, hotel, accommodationFee, date and invoice. Moreover, physicalLocation has four sub-concepts (viz., city, province, region and state), date has two sub-concepts (viz., beginDate and lastDate), and invoice has a single sub-concept (viz., hotelInvoice). Figure 3.8 shows
the dependency hypergraph after inserting the hotel ontology. Note that six semantic correspondences among the concepts of the hotel and the event ontologies have been determined by the SemFiT service (viz., by the getEquivalentConcepts method of SemFiT). Moreover, a semantic correspondence has been established between two concepts defined in the hotel ontology and the e-commerce ontology, respectively.

Figure 3.7: A view of the hypergraph after inserting the ConferenceService.

Figure 3.8: A view of the hypergraph after inserting the hotel ontology.

It is worth observing that some of the semantic correspondences found by SemFiT are due to the similarity between the syntax of the compared concepts, such as the relation between the invoice concept of the hotel ontology and the invoice concept of the e-commerce ontology, the relation between the city concept of the hotel ontology and the city concept of the event ontology, and the relation between
the physicalLocation concept of the hotel ontology and the location concept of the event ontology. Such semantic correspondences are represented in the hypergraph by the following three $E_\equiv$ hyperedges: ($\{\text{invoice}\},\{\text{invoice}\},0.82$), ($\{\text{city}\},\{\text{city}\},0.82$) and ($\{\text{physicalLocation}\},\{\text{location}\},0.57$).

Other correspondences are instead the result of applying searches in WordNet, such as the relation between the country concept of the event ontology and the state concept of the hotel ontology, between the startDate concept of the event ontology and the beginDate of the hotel ontology, and between the endDate concept of the event ontology and the lastDate concept of the hotel concept. In the latter two cases, the similarity does not derive from direct synonyms in WordNet, but from the tokens that compose the words (viz., start and begin, as well as, end and last). These semantic correspondences consist of the following $E_\equiv$ hyperedges: ($\{\text{country}\},\{\text{state}\},0.82$), ($\{\text{startDate}\},\{\text{beginDate}\},0.82$) and ($\{\text{endDate}\},\{\text{lastDate}\},0.82$).

Finally, the dateTime concept of the event ontology and the date concept of the hotel ontology result equivalent because their syntax and their structure, indeed both have children that have same similarity. Such a correspondence corresponds to the $E_\equiv$ hyperedge ($\{\text{dateTime}\},\{\text{date}\},0.59$).

![Hypergraph View](image)

Figure 3.9: A view of the hypergraph after inserting the HotelService.

Next, AddService invokes the MakeProfiles function in order to compute the intra-service dependencies of the HotelService. As one can note in Figure 2.6, the root composite process of HotelService is a repeat-until process. Hence, to determine the possible behaviour of HotelService, MakeProfiles invokes the Alt function (Figure 3.3) over the child process choice#2 of the repeat-until root process. Alt(choice#2) returns two, indeed, choice#2 has to be executed twice in order to yield all the outputs producible by its children. Hence, MakeProfiles(repeat-until(choice#2)) ex-
pands in `MakeProfiles(choice(choice#2,sequence(choice#2,choice#2)))`, which returns three distinct profiles of `HotelService`, that we name $H_1$, $H_2$, and $H_3$. In particular, $H_1$ corresponds to executing the process `{collectInfoHotel}`, $H_2$ corresponds to executing the processes `{selectHotel, payment}`, and $H_3$ corresponds to executing the processes `{collectInfoHotel, selectHotel, payment}`. As one can observe in Figure 3.9, `HotelService` is synthesised in the dependency hypergraph by three $E_S$ hyperedges, namely, ({city, state}, {infoHotel}, $H_1$), ({hotel, beginDate, lastDate, creditCard, contactInformation}, {accomodationFee, hotelInvoice}, $H_2$), and ({city, state, hotel, beginDate, lastDate, creditCard, contactInformation}, {infoHotel, accomodationFee, hotelInvoice}, $H_3$).

We add now the `RatingOne` service to the local registry. As one can note in Figure 3.10, the process model of `RatingOne` employs concepts defined in the `bankOntology`. Figure 3.10 depicts the dependency hypergraph after inserting the `bankOntology`. Note that for the sake of simplicity, Figure 3.10 shows only the concepts of `bankOntology`, since there exists no semantic correspondences among the concepts of `bankOntology` and the concepts of the previously inserted ontologies.

As one can note in Figure 2.4, the `RatingOne` service has a single possible behaviour, since it has to necessarily execute all its atomic processes. Indeed, `RatingOne` has a root composite process of type `sequence`, and an internal composite process of type `split+join`. As described in Subsection 2.1.1, each child process of a `sequence` or `split+join` composite process has to be executed. Figure 3.11 illustrates the dependency hypergraph after inserting `RatingOne`. In particular, the single profile $RO_1$ of `RatingOne` is synthesised by the $E_S$ hyperedge which links the set of (input) concepts \{amountOfCredit, balance, guarantee, firstRating, secondRating, thirdRating\} to the set of (output) concepts \{firstRating, secondRating, thirdRating, rating\}.

Finally, we conclude our example by adding to the local registry the `RatingTwo` service, depicted in Figure 2.5. As `RatingOne`, `RatingTwo` employs only concepts de-
3.2 Satisfying the functional requirements of a client query

As described in Subsection 1.3.3, the discovery technique presented in this thesis consists of two main parts, namely, a functional analysis, which selects (sets of) services with respect to their functional attributes, and a behavioural analysis, which checks whether the (sets of) services returned by the functional analysis satisfy some behavioural properties.

The functional analysis takes as input the functional requirements of a client
query (viz., a list of ontology-annotated inputs and outputs of the service to be found) and it returns as output those sets of services which satisfy the given requirements. Before presenting the algorithm which summarises the functional analysis, we formally define when a set of services satisfies the functional requirements of a client query. Let $\text{In}(s), \text{Out}(s)$ denote the inputs and the outputs of a service $s$, respectively. Similarly, let $\text{In}(Q), \text{Out}(Q)$ denote the inputs and the outputs of a client query.

**Definition 3.7 (functional requirements).** A set of services $S$ satisfies the functional requirements of a query $Q$ if and only if (1) every query output $o$ subsumes a concept $x$ produced by some service in $S$ (viz., $x$ is a sub-concept of $o$), and (2) every service input $i$ subsumes either a query input or an output $x$ of some service in $S$ (viz., $x$ is a sub-concept of $i$).

Namely, we say that a set of services $S$ satisfies the functional requirements of a client query $Q$ if and only if (1) every query output $o$ subsumes a concept $x$ produced by some service in $S$ (viz., $x$ is a sub-concept of $o$), and (2) every service input $i$ subsumes either a query input or an output $x$ of some service in $S$ (viz., $x$ is a sub-concept of $i$).

Note that Definition 3.7 refers to the functional attributes of services and it does not consider service behaviour, which will be considered by the next step of the discovery technique (viz., the behavioural analysis). Therefore, a set of services that satisfies a query according to Definition 3.7 may possibly lock during their interaction. Yet, in the rest of the Section – when clear from the context – we shall simply say that a service “satisfies the query” instead of “satisfies the functional requirements of the query”.

![Diagram](image_url)
3.2. SATISFYING THE FUNCTIONAL REQUIREMENTS OF A CLIENT QUERY

We present the algorithm for discovering sets of services in the next Subsection and we show its ability of finding minimal sets of services in Subsection 3.2.2. Soundness and completeness of the algorithm are discussed in Subsection 3.2.3.

3.2.1 Discovering sets of services

As previously discussed in Section 3.1, the functional dependencies among and within services are collected into a hypergraph. As one may expect, hence, the functional analysis consists of a visit of the dependency hypergraph. The behaviour of the functional analysis is summarised by the recursive function \text{FunAn}, whose pseudo-code is listed in Figure 3.13.

\begin{verbatim}
1. \text{FunAn}(hypergraph } H, \text{ query } Q, \text{ set composition, set needed, set available})
2. \hspace{1em} if (needed = \emptyset) then return composition
3. \hspace{1em} else
4. \hspace{2em} out = extract(needed);
5. \hspace{2em} S = \{s \mid \text{out} \in \text{Out}(s) \lor (\exists c \in \text{Out}(s) \mid c \preceq \text{out})\};
6. \hspace{2em} if (S = \emptyset) then fail
7. \hspace{2em} else
8. \hspace{3em} forall service \ s \ in \ S \ do
9. \hspace{4em} composition' = composition \cup \{s\};
10. \hspace{3em} forall service \ t \ in \ composition \ do
11. \hspace{4em} R = composition' \setminus \{t\};
12. \hspace{4em} if (\nexists x \in \text{Out}(t) : \exists o \in \text{Out}(Q) : (x \preceq o \land \nexists z \in \bigcup_{r \in R} \text{Out}(r) : z \preceq o) \lor
13. \hspace{4em} \exists i \in \bigcup_{r \in R} \text{In}(r) : (x \preceq i \land \nexists z \in \bigcup_{r \in R} (\text{Out}(r) \cup \text{In}(Q)) : z \preceq i))
14. \hspace{4em} \text{then fail};
15. \hspace{4em} available' = available \cup \text{Out}(s);
16. \hspace{4em} needed' = \{x \mid x \in (\text{needed} \cup \{y \mid y \in \text{In}(s) \land \nexists z \in \text{In}(Q) : z \preceq y\})
17. \hspace{4em} \land \nexists a \in \text{available'} : a \preceq x\};
18. \hspace{3em} \text{FunAn}(H, Q, composition', needed', available');
\end{verbatim}

Figure 3.13: Pseudo-code summarising the functional analysis.

In the pseudo-code of Figure 3.13, \text{In}(s) denotes the inputs of the service \ s, that is, with respect to the dependency hypergraph defined in Subsection 3.1.1, the set \ \{i \mid \exists I, O : (I, O, s) \in E_S \land i \in I\} \ while \ \text{Out}(s) \ denotes \ the \ outputs \ of \ \ s, \ namely \ \{o \mid \exists I, O : (I, O, s) \in E_S \land o \in O\}.

The \text{FunAn} function inputs five parameters: the hypergraph \ H, the client query \ Q, the set \text{composition} of the services selected so far (initially empty), the set of the \text{needed} outputs to be generated (initially the query outputs), and the set of the \text{available} outputs (initially empty).
FunAn explores the hypergraph starting from those nodes corresponding to the query outputs, and it continues by visiting backwards the hyperedges until reaching, if possible, the query inputs. As we will see in Subsection 3.2.2, FunAn determines all the minimal sets of services that satisfy the client query.

If there are no (more) outputs to be generated \((\text{needed} = \emptyset)\), FunAn returns the set \(\text{composition}\) of the services found (line 2). Otherwise (line 3) FunAn withdraws an output \(\text{out}\) from the set of \(\text{needed}\) output (line 4) and computes the set \(S\) of the services which produce (a \subConcept\ of) \(\text{out}\) (line 5). If \(S\) is empty, that is, \(\text{out}\) can not be generated by any service in the local registry, then FunAn fails (line 6) since the query can not be fulfilled. Otherwise (line 7) for each service \(s\) which generates (a \subConcept\ of) \(\text{out}\) (line 8), FunAn adds \(s\) to \(\text{composition}\) (line 9), updates \(\text{available}\) by adding the outputs of \(s\) (line 16), and updates \(\text{needed}\) by adding the inputs of \(s\) and by removing the concepts that are now available (lines 17–18). Next, FunAn continues recursively (line 19).

It is worth noting that, by exploring \(\text{profiles}\), FunAn addresses the discovery of sets of services, while by employing the extended notion of \subConcept\ relationships (Definition 3.4) it deals with (different) ontologies.

In the next Subsection, we discuss the loop at lines 10–15 of the FunAn pseudo-code (Figure 3.13) whose role is to discard (by failing) all the non-minimal sets of services that satisfy the query.

3.2.2 Discovering \textit{minimal} sets of services

Let us now consider the remaining lines 10–15 of the pseudo-code of FunAn. As already anticipated, the role of this loop is to discard (by failing) any non-minimal set of services that satisfies the query.

Let us first formalise the (obvious) notion of minimality.

\textbf{Definition 3.8 (minimal set of services).} Let \(Q\) be a query and let \(S\) be a set of services that satisfies the functional requirements of \(Q\). \(S\) is minimal \(\iff\) \(\nexists S' \subset S : S' \text{ satisfies the functional requirements of } Q\).

To illustrate the nature of non-minimal sets of services, consider the following simple example. Consider a query taking as input \(e\) and requiring as outputs \(a, b, c\) and \(d\), and three services \(s_1, s_2\) and \(s_3\) (Figure 3.14). Service \(s_1\) takes as input \(e\) and returns as output \(a, b\) and \(c\), service \(s_2\) takes as input \(e\) and produces as output \(a\) and \(d\), while service \(s_3\) takes as input \(e\) and returns as output \(b\) and \(d\). The set of services consisting of \(s_1, s_2\) and \(s_3\) satisfies the query but it is not minimal because the outputs produced by \(s_2\) are contained in the set of outputs produced by \(s_1\) and \(s_3\) (viz., \(\{a, d\} \subset \{a, b, c, d\}\)). On the other hand, both \(\{s_1, s_3\}\) and \(\{s_1, s_2\}\) are minimal sets of services that satisfy the query.

Intuitively speaking, the loop at lines 10–15 checks whether the inclusion of the new service \(s\) in the set of services \(\text{composition}\) makes some other service in
3.2. SATISFYING THE FUNCTIONAL REQUIREMENTS OF A CLIENT QUERY

Figure 3.14: Example of minimal sets of services.

composition not strictly necessary to satisfy the query. Note that $s$ is certainly needed to satisfy the query, since the set $\text{composition} \setminus \{s\}$ is not able to produce $\text{out}$ (line 5). Note, indeed, that $\text{FunAn}$ does not analyse the service $s$ (line 10).

$\text{FunAn}$ hence checks, for each service $t$ in $\text{composition}$ (line 10), that the condition at lines 12–14 does not hold. Such condition holds if all the “useful” outputs produced by a service $t$ are already produced by the other services in $\text{composition} \cup \{s\} \setminus \{t\}$. An output of $t$ is considered useful if it is a subConcept of a query output or of an input needed by some other service (and not part of the query inputs). Therefore, if the condition at lines 12–14 holds, this means that the inclusion of $s$ in the set of services has made service $t$ not strictly necessary to achieve the goal. If this is the case, $\text{FunAn}$ fails (line 15) to avoid constructing non-minimal sets of services.

It is worth noting that, while the condition at lines 12–14 is verbose, its verification simply reduces to a few trivial operations over (small sized) sets of data.

Finally, we prove that the condition employed at lines 12–14 is both necessary and sufficient to establish the minimality of a set of services that satisfies a query.

**Property 3.1** Let $Q$ be a query and let $S$ be a set of services that satisfies the functional requirements of $Q$.

$S$ is minimal $\iff \forall t \in S, \exists x \in \text{Out}(t) :$

1. $\exists o \in \text{Out}(Q) : (x \preceq o \land \ \exists z \in \bigcup_{r \in S \setminus \{t\}} \text{Out}(r) : z \preceq o)$

or

2. $\exists i \in \bigcup_{r \in S \setminus \{t\}} \text{In}(r) : (x \preceq i \land \ \exists z \in \bigcup_{r \in S \setminus \{t\}} \text{Out}(r) \cup \text{In}(Q) : z \preceq i)$.

**Proof.** We write $S \models Q$ to denote that $S$ satisfies the functional requirements of $Q$.

$(\Rightarrow)$

$S$ is minimal

$\iff \{\text{by Definition 3.8 and by Definition 3.7}\}$

$(\Leftarrow)$

$S$ is minimal

$\iff \{\text{by Definition 3.8 and by Definition 3.7}\}$
∀t ∈ S :

(∃o ∈ Out(Q) : ∃z ∈ \bigcup_{r \in S \setminus \{t\}} Out(r) : z ≤ o)
∧
(∃i ∈ \bigcup_{r \in S \setminus \{t\}} In(r) : ∃z ∈ (\bigcup_{r \in S \setminus \{t\}} Out(r) \cup \ln(Q)) : z ≤ i)

⇒ \{ \text{since } S \models Q, \text{ by Definition 3.7} \}

∀t ∈ S \: \exists x ∈ Out(t) :

(∃o ∈ Out(Q) : x ≤ o \land ∃z ∈ \bigcup_{r \in S \setminus \{t\}} Out(r) : z ≤ o)
∧
(∃i ∈ \bigcup_{r \in S \setminus \{t\}} In(r) : x ≤ i \land ∃z ∈ (\bigcup_{r \in S \setminus \{t\}} Out(r) \cup \ln(Q)) : z ≤ i)

(⇐⇐) Let us now suppose that S is not minimal. Then,

S is not minimal

⇒ \{ \text{since } S \models Q, \text{ by Definition 3.7 and by Definition 3.8} \}

∃t ∈ S :

(\forall o ∈ Out(Q) \: \exists z ∈ \bigcup_{r \in S \setminus \{t\}} Out(r) : z ≤ o)
∧
(\forall i ∈ \bigcup_{r \in S \setminus \{t\}} In(r) \: \exists z ∈ (\bigcup_{r \in S \setminus \{t\}} Out(r) \cup \ln(Q)) : z ≤ i)

⇒ \{ \text{since } \exists t \in S(\forall a ∈ A(∃z ∈ B : P(t, a, z)) \Rightarrow \exists t \in S(\exists a ∈ A(∃z ∈ B : P(t, a, z)))} \}

∃t ∈ S :

(\exists o ∈ Out(Q) : \exists z ∈ \bigcup_{r \in S \setminus \{t\}} Out(r) : z ≤ o)
∧
(\exists i ∈ \bigcup_{r \in S \setminus \{t\}} In(r) : \exists z ∈ (\bigcup_{r \in S \setminus \{t\}} Out(r) \cup \ln(Q)) : z ≤ i)

□

Property 3.1 hence ensures the minimality of the sets of services determined by the FunAn function.

3.2.3 Soundness, completeness and complexity

We conclude this Section 3.2 with a discussion of the soundness and completeness of the functional analysis, whose behaviour is summarised in Figure 3.13, where the pseudo-code of the FunAn function is listed. A worst-case analysis of the time complexity of the functional analysis is also presented.
3.2. SATISFYING THE FUNCTIONAL REQUIREMENTS OF A CLIENT QUERY

Soundness

As detailed in Subsection 3.2.1, each instance of the functional analysis returns a set of services which satisfies the functional requirements of the query. We have formally defined when a set of services satisfies a query at the beginning of this Section (Definition 3.7 of Section 3.2). The following proposition establishes the soundness of the functional analysis, namely, that each set of services returned by the FunAn function indeed satisfies the functional requirements of the query as per Definition 3.7.

Proposition 3.1 Let $S$ be a set of services returned by the functional analysis (viz., by the FunAn function) for a given query $Q$. Then, $S$ satisfies the functional requirements of $Q$.

Proof. The proof is organised in three steps. First, we establish an invariant property $\text{Inv}$ that holds for every invocation of FunAn. Then, we prove that $\text{Inv}$ implies the desired property (viz., $S$ satisfies $Q$) when FunAn terminates. Finally, we show that FunAn always terminates.

(1) We note that the set $\text{needed}$ – which initially contains the query outputs – is updated whenever a new service $s$ is added to $\text{composition}$. Then, FunAn adds to $\text{needed}$ the inputs of $s$ which neither belong to the query inputs nor to the outputs of the services in $\text{composition}$, while the concepts which subsume the outputs of $s$ are removed from $\text{needed}$. Hence we observe that whenever a recursive call $\text{FunAn}(H, Q, \text{composition}, \text{needed}, \text{available})$ is issued, the following invariant property $\text{Inv}$ holds:

$$\text{needed} = \{x : x \in (\text{Out}(Q) \cup \{u \mid u \in \bigcup_{t \in S} \text{In}(t) \land \nexists v \in \text{In}(Q) : v \leq u}) \land \nexists y \in \bigcup_{t \in S} \text{Out}(t) : y \leq x\}$$

where $S$ denotes the set of services selected so far (viz., $\text{composition}$).

(2) We observe that FunAn returns the set $\text{composition}$ when $\text{needed} = \emptyset$. Then,

$$\text{needed} = \emptyset$$

$\Rightarrow$ {by $\text{Inv}$}

$$\nexists x : x \in (\text{Out}(Q) \cup \{u \mid u \in \bigcup_{t \in S} \text{In}(t) \land \nexists v \in \text{In}(Q) : v \leq u}) \land \nexists y \in \bigcup_{t \in S} \text{Out}(t) : y \leq x$$

$\Rightarrow$ {since $\nexists x : (x \in A \cup B \land \nexists y \in C : P(x, y)) \Rightarrow \forall x : (x \in A \Rightarrow \exists y \in C : P(x, y)) \land (x \in B \Rightarrow \exists y \in C : P(x, y))$}
∀x : x ∈ Out(Q) ⇒ ∃y ∈ \bigcup_{t \in S} Out(t) : y ≤ x
∧
(x ∈ \bigcup_{t \in S} In(t) \land \exists v \in In(Q) : v ≤ u) ⇒ ∃y ∈ \bigcup_{t \in S} Out(t) : y ≤ x
⇒ \{\text{since } (A \land \neg B) \Rightarrow C \text{ implies } A \Rightarrow (B \lor C)\}
∀x : x ∈ Out(Q) ⇒ ∃y ∈ \bigcup_{t \in S} Out(t) : y ≤ x
∧
x ∈ \bigcup_{t \in S} In(t) ⇒ ∃y ∈ (\bigcup_{t \in S} Out(t) \cup In(Q)) : y ≤ x
⇒ \{\text{by Definition 3.7}\}

S satisfies the functional requirements of Q

Hence the invariant property Inv guarantees that when needed = \emptyset then composition is a set of services that satisfies the functional requirements of the query.

(3) Furthermore, FunAn always terminates. We first observe that if the set of those services which produce (a sub-concept of) out is not empty (line 5, Figure 3.13), then none of the services in such a set is already part of composition (otherwise out would not still belong to needed). Hence, each available service can be inserted into composition only once.

Let candidates be the set of the candidate services which can be inserted to composition. Then, we define the terminating function \(T = |\text{candidates}|\). Initially, the candidates set consists of all the available services in the local registry. Next, at each recursive invocation, FunAn selects a service \(s\) from candidates and inserts it to composition. Since \(s\) can be inserted to composition only once, candidates = candidates \\{s\}, and \(T\) decreases accordingly. Hence, FunAn terminates at most after inserting all the candidate services into composition, namely, when \(T = 0\). In particular, the last recursive call of FunAn either succeeds, if needed is empty, or it fails, otherwise (since each out ∈ needed can not be produced by any of the available services).

\(\square\)

The result below immediately follows from Proposition 3.1 and Property 3.1.

**Corollary 3.1** Let \(S\) be a set of services returned by the functional analysis for a given query \(Q\). Then, \(S\) is minimal and satisfies the functional requirements of \(Q\).

**Completeness**

We proved that each set of services returned by the functional analysis satisfies Definition 3.7. Now, we show hereafter that the functional analysis returns *all* the minimal set of services which satisfy a given query.
3.2. SATISFYING THE FUNCTIONAL REQUIREMENTS OF A CLIENT QUERY

Proposition 3.2 Let \( S = \{P_1, ..., P_n\} \) be a minimal set of services such that \( S \) satisfies the functional requirements of a given query \( Q \). If there exists such a set of services \( S \), then \( S \) is returned by the functional analysis.

Proof. Suppose by contradiction that there exists a minimal set of services \( S = \{P_1, ..., P_n\} \) such that \( S \) satisfies \( Q \), and \( S \) is not returned by the functional analysis. There are two possible reasons why the functional analysis does not return \( S \).

(1) There exists some service \( P_i \in S \) which is not selected by the functional analysis. This means that none of the recursive calls of the functional analysis extracts from the set of the concepts to be generated (viz., the needed set) a concept \( \text{out}_i \) produced by \( P_i \). This is possible in the following two cases:

- \( P_i \) generates no concepts useful to satisfy \( Q \). Namely, \( P_i \) produces neither concepts which belong to the query outputs, nor concepts taken as input by some \( P_j \in S \ (P_j \neq P_i) \). Hence, \( P_i \) is completely useless to satisfy \( Q \). Thus, we obtain a contradiction, since the set of services \( S \) is not minimal.

- \( P_i \) generates a concept \( c \) useful to satisfy \( Q \), yet, \( c \) is produced also by a service \( P_j \in S \), with \( P_j \neq P_i \). In particular, suppose that, at some step \( n \), the functional analysis extracts from the set of the concepts to be generated (viz., the needed set) a concept \( \text{out}_j \) (with \( \text{out}_j \neq c \)) which is produced by \( P_j \). Hence, \( P_j \) is added to \( S \), and next, the outputs of \( P_j \) are removed from the needed set and added to the set of the available concepts (viz., the available set). Hence, also the concept \( c \) is removed from the needed set (if present) and added to the available set. Consequently, the concept \( c \) will be never extracted from the needed set, so that \( P_i \) will not be selected by the functional analysis. Yet, \( P_i \) is not strictly necessary to satisfy \( Q \), hence, we obtain a contradiction, since the set of services \( S \) is not minimal.

(2) The set \( S = \{P_1, ..., P_n\} \) (or a subset of \( S \)) is discarded by the functional analysis. This is the case of a service \( P_i \in S \) which generates \( c \) and of a service \( P_j \in S \ (P_i \neq P_j) \) which generates \( \text{out}_j \) and \( c \). Suppose that at some step \( n \), the functional analysis extracts \( c \) from the needed set and selects \( P_i \). Yet, when at some step \( m > n \), \( \text{out}_j \) is extracted from needed, the functional analysis selects \( P_j \), thus making \( P_i \) useless, since \( c \) is also produced by \( P_j \) (Property 3.1 does not hold). We obtain hence a contradiction, since the set of services \( S \) is not minimal.

Hence, the functional analysis returns all the minimal set of services which satisfy the functional requirements of a given query \( Q \).
CHAPTER 3. FUNCTIONAL ANALYSIS

Complexity analysis

Finally, let us sketch the worst-case analysis of the time complexity $T_{\text{FunAn}}(P)$ of executing $\text{FunAn}$ on a dependency hypergraph containing $|P|$ profiles. $T_{\text{FunAn}}(P)$ derives from the time needed to perform all the instances generated by recursively executing the $\text{FunAn}$ function. Intuitively, the first instance of $\text{FunAn}$ extracts a concept $\text{out}_1$ from the needed set. At most, there exist $|P|$ profiles which produce (a subConcept of) $\text{out}_1$. Hence, $\text{FunAn}$ splits in $|P|$ instances. Then, consider, for example, the instance $i$, with composition $\{p_i\}$. Such an instance, in turn, extracts a concept $\text{out}_2$ from the needed set. At most, there exist $|P| - 1$ profiles (viz., the set of profiles $P \setminus \{p_i\}$) which produce (a subConcept of) $\text{out}_2$. Hence, $\text{FunAn}$ splits in $|P| - 1$ instances. Next, consider, for example, an instance $j$, with composition $\{p_i, p_j\}$. Such an instance $j$ extracts a concept $\text{out}_3$ from the needed set. Similarly, there exist $|P| - 2$ profiles which produce (a subConcept of) $\text{out}_3$ (viz., the set of profiles $P \setminus \{p_i, p_j\}$). Thus, $\text{FunAn}$ splits in $|P| - 2$ instances, and so on. Therefore, $\text{FunAn}$ generates $|P| \times (|P| - 1) \times (|P| - 2) \times \ldots \times 1$ instances. The minimality check (lines 10–15 in Figure 3.13) is the most expensive task of an instance, by executing $O(|P|^2)$ steps. Then, each instance of the $\text{FunAn}$ function belongs to $O(|P|^3)$, since it executes the minimality check for each profile producing the out concept extracted from the needed set. Hence, the $\text{FunAn}$ function requires exponential time, namely $T_{\text{FunAn}}(P) \in NP$. Finally, it is worth noting that $|P| \in O(|S|)$, where $S$ denotes the set of services contained in the local service registry. We can hence write $T_{\text{FunAn}}(S) \in NP$.

3.2.4 An example

In Subsection 3.1.3, we presented a sample dependency hypergraph – illustrated in Figures 3.9 and 3.12 – which results from inserting four services (viz., ConferenceService, HotelService, RatingOne, RatingTwo) and four ontologies (viz., e-commerce, event, hotel, bankOntology) to the local service registry. We continue hereafter the example introduced in Subsection 3.1.3. First, we present two client queries, and next, we show how the $\text{FunAn}$ function (Figure 3.13), previously described in Subsection 3.2.1, explores the hypergraph in order to determine those sets of services capable of satisfying the functional requirements of the client queries.

We consider first a client wishing to plan its participation in an international conference by registering to the conference and by booking her hotel accommodation, and receiving the conference and hotel confirmations. The client query may be composed by the following parameters:

- **inputs**: internationalConference, contactInformation, creditCard, hotel, and

- **outputs**: registrationReceipt, invoice.
3.2. SATISFYING THE FUNCTIONAL REQUIREMENTS OF A CLIENT QUERY

where the concepts contactInformation, creditCard, registrationReceipt and invoice belong to the e-commerce ontology, the internationalConference belongs to the event ontology, and the hotel concept belongs to the hotel ontology.

Hence, this first sample query specifies only the functional attributes (i.e., inputs and outputs) of the service to be found, namely, such a query is a so-called functional query. It is worth to anticipate here that, as a consequence, the next step of the discovery technique described in this thesis, i.e., the behavioural analysis, will check, for each set of services found by the FunAn function, only whether the services in the set can be really composed together without dead-locking.

Let us now describe how FunAn processes the first sample client query. As detailed in Figure 3.13, FunAn takes as input five parameters, in particular, in this example: the hypergraph built in Subsection 3.1.3, the client query specified above, the set available of the available concepts (initially \{internationalConference, contactInformation, creditCard, hotel\}), the set needed of the concepts to be produced (initially \{registrationReceipt, invoice\}), while composition is an empty set.

FunAn starts by withdrawing a concept from the needed set. Let us suppose that FunAn withdraws first the invoice concept. Next, FunAn searches for those ES hyperedges that include (a subConcept of) invoice in their heads, namely, it searches for those service profiles which produce (a subConcept of) invoice. As one can note in Figure 3.9, there exist two profiles which produce the invoice concept, i.e., the profiles \(H_2\) and \(H_3\) of HotelService. Indeed they both produce hotelInvoice, which is a direct sub-concept of invoice (i.e., there is an \(E_S\) hyperedge which connects hotelInvoice to invoice). Then, FunAn splits in two separate instances, the first adding \(H_2\) to composition, and the second adding \(H_3\) to composition.

Consider composition \(\{H_2\}\). \(H_2\) has five inputs, namely, hotel, contactInformation, creditCard, beginDate, and lastDate. While hotel, contactInformation, creditCard belong to the query inputs, there is no query input which is a subConcept of beginDate or lastDate. Hence, FunAn adds both beginDate and lastDate to the needed set. Furthermore, FunAn adds the outputs of \(H_2\) (viz., accomodationFee and hotelInvoice) to the available set. Then, FunAn continues recursively. Let us now suppose that FunAn withdraws the registrationReceipt concept from the needed set. The dependency hypergraph contains two profiles producing registrationReceipt, namely, \(C_1\) and \(C_2\), which synthetise two possible behaviour of ConferenceService. Again, FunAn splits in two instances, the first associated with the composition \(\{H_2, C_1\}\), and the second associated to the composition \(\{H_2, C_2\}\).

Consider the first instance. FunAn adds the outputs of the profile \(C_1\) (viz., registrationFee, city, country, startDate, endDate, registrationReceipt) to the available set. The needed set is now empty. Indeed, the now available concepts startDate and endDate are equivalent concepts of beginDate and lastDate, which are hence removed from the needed set. Moreover, the available set contains the inputs of the profile \(C_1\) (viz., academicEvent, contactInformation and creditCard), which have not to be added to the needed set. Hence, \(\{H_2, C_1\}\) is a successfull composition, since it satisfies the functional requirements of the client query.
Consider now the second instance, corresponding to the composition \( \{H_2, C_2\} \). FunAn adds the outputs of the profile \( C_2 \) (viz., \( \text{registrationReceipt} \), \( \text{city} \), \( \text{country} \), \( \text{startDate} \), \( \text{endDate} \), \( \text{registrationReceipt} \)) to the available set. Then, it removes \( \text{beginDate} \) and \( \text{endDate} \) from the needed set, as they are now available. Yet, FunAn adds the \( \text{bankAccount} \) concept to needed, indeed \( \text{bankAccount} \) is the only inputs of \( C_2 \) which is not available. Next, FunAn continues recursively. FunAn withdraws \( \text{bankAccount} \) from needed, yet, there are no profiles which produce (a subConcept of) \( \text{bankAccount} \). Hence, this instance of FunAn terminates by failing.

We consider now the instance associated to composition \( \{H_3\} \). \( H_3 \) has seven inputs, namely, \( \text{city} \), \( \text{contactInformation} \), \( \text{state} \), \( \text{hotel} \), \( \text{creditCard} \), \( \text{beginDate} \), and \( \text{lastDate} \), and produces three outputs, viz., \( \text{infoHotel} \), \( \text{accommodationFee} \) and \( \text{hotelInvoice} \). FunAn adds both \( \text{beginDate} \) and \( \text{lastDate} \) to the needed set, since they do not belong to the query inputs, and it adds \( \text{infoHotel} \), \( \text{accommodationFee} \) and \( \text{hotelInvoice} \) to the available set. Then, FunAn continues recursively. Let us now suppose that FunAn withdraws the \( \text{registrationReceipt} \) concept from the needed set, which is produced by the profiles \( C_1 \) and \( C_2 \) of ConferenceService. FunAn splits in two instances, corresponding to the compositions \( \{H_3, C_1\} \) and \( \{H_3, C_2\} \), respectively.

Consider the former instance. FunAn adds the outputs of \( C_1 \) (viz., \( \text{registrationReceipt} \), \( \text{city} \), \( \text{country} \), \( \text{startDate} \), \( \text{endDate} \), \( \text{registrationReceipt} \)) to the available set, and it next removes \( \text{beginDate} \) and \( \text{lastDate} \) from the needed set. Then, no input of \( C_1 \) has to be added to needed, which is now empty. Hence, \( \{H_3, C_1\} \) is a successful composition, as well.

Finally, we consider the instance associated to the composition \( \{H_3, C_2\} \). FunAn adds the outputs of \( C_2 \) (viz., \( \text{registrationReceipt} \), \( \text{city} \), \( \text{country} \), \( \text{startDate} \), \( \text{endDate} \), \( \text{registrationReceipt} \)) to the available set, and it next removes \( \text{beginDate} \) and \( \text{lastDate} \) from the needed set. Then, FunAn adds the input \( \text{bankAccount} \) of \( C_2 \) to needed, as it is not available. Next, FunAn withdraws \( \text{bankAccount} \) from needed, yet, there are no profiles which produce (a subConcept of) \( \text{bankAccount} \). Hence, this instance of FunAn terminates by failing.

Summing up, FunAn returns two successful compositions, namely, \( \{H_2, C_1\} \) and \( \{H_3, C_1\} \). Each of them will be next analysed by the behavioural analysis.

We consider now a particular scenario, where the provider of the CreditPortal service – previously described in Subsection 2.1.2 and illustrated in Figure 2.3 – wants to enhance its service and hence decides to externalise the CreditPortal section which computes customer rating (viz., the sub-tree with root \text{sequence}\#10 in Figure 2.3). The CreditPortal provider, hence, wants to search for a service which first takes as input the client data (viz., the concepts \text{amountOfCredit} , \text{guarantee} , \text{balance} of the bankOntology), and next returns as output the client rating (viz., the \text{rating} concept of the bankOntology). Furthermore, the CreditPortal provider wants to search for a service that can safely substitute such a portion of CreditPortal, without altering the behaviour of the complete CreditPortal application. The CreditPortal provider specifies the inputs, the outputs and the expected behaviour of the desired
service by means of a so-called *behavioural query*, which can be expressed with an OWL-S process model, such as the one in Figure 3.15. In this case, the FunAn function returns all the sets of services which satisfy the functional requirements of the query, while the behavioural analysis check, for each set, whether the composition of the services in the set features a behaviour equivalent to the one expressed by the behavioural requirements of the query.

![Figure 3.15: An OWL-S process model representing a behavioural client query.](image)

As anticipated in the Introduction, a behavioural query is split into two parts, which express the functional requirements and the behavioural requirements of the query, respectively. In particular, the functional requirements of the query depicted in Figure 3.15 are expressed by the following parameters:

- **inputs**: amountOfCredit, guarantee, balance, and
- **output**: rating.

We describe hereafter the behaviour of the FunAn function, while a complete discussion of the behavioural analysis can be found in the next Chapter. FunAn takes as input the dependency hypergraph built in Subsection 3.1.3, the client query specified above, the set available of the available concepts (initially \{amountOfCredit, guarantee, balance\}), the set needed of the concepts to be produced (initially \{rating\}), and the empty set composition. FunAn withdraws the rating concept from the needed set, and searches for those profiles which produce (a subConcept of) rating. As one can note in Figure 3.12, both the single profile RO\(_1\) of the RatingOne service and the three profiles RT\(_1\), RT\(_2\) and RT\(_3\) of the RatingTwo service produce as output the rating concept. FunAn creates a new instance for each of such profiles.

Consider the instance which adds RO\(_1\) to the set composition. FunAn adds the outputs of RO\(_1\) (viz., rating, firstRating, secondRating and thirdRating) to the available set. Next, since all the inputs of RO\(_1\) (viz., amountOfCredit, guarantee, balance, firstRating, secondRating and thirdRating) belong to the available set, the needed set stays empty, and FunAn terminates by returning the successful composition \{RO\(_1\)\}. Similarly, instances which add RT\(_1\), RT\(_2\) and RT\(_3\), respectively, to the composition set, terminate by returning the successful compositions \{RT\(_1\)\}, \{RT\(_2\)\} and \{RT\(_3\)\}. 
3.3 Discussion

In this Chapter we described the functional analysis of our discovery framework. The functional analysis is capable of determining functional dependencies among and within services, which are synthesised into a dependency hypergraph. In particular, the nodes of the hypergraph represent concepts of the ontologies employed to describe the functional attributes of services, while hyperedges represent relationships among such concepts. Besides hyperedges representing subConcept and equivalentConcept relationships among concepts, there are also hyperedges representing functional dependencies within services, viz., the so-called profiles. Namely, a profile is the functional characterisation of a particular behaviour of a service. Furthermore, dependencies among different services are natively modelled by the hypergraph. There is a dependency between two services, if there exists a (concept) node which belongs to the head of the first service profile and to the tail of the second service profile. The functional analysis also provides a suitable matchmaking algorithm that, given a client request specifying the (ontology-annotated) inputs and the outputs of the desired service, returns all the sets of services capable of satisfying the (functional) request. In Subsection 3.2.3 we proved that such an algorithm is correct, complete and always terminates.

A relevant distinction with respect to the state-of-the-art approaches such as, e.g., [66, 41, 7], is the employed notion of concept compatibility. In Subsection 3.1.1, we argued that a concept \( c \) matches a concept \( d \) if either \( c \) is (semantically) equivalent to \( d \), or \( c \) is a sub-concept of \( d \). The approaches in [66, 41, 7], instead, also consider the possibility that \( c \) matches \( d \) if \( d \) is a sub-concept of \( c \). Yet, in our opinion this violates the OWL-S specification [64], which states that two processes are type compatible if for each output of one that flows to the input of the other, the type of the output is a subtype of the type of the input. Hence, for example, a process which outputs a car concept is compatible with a process which inputs a vehicle concept, while, vice versa, a process which outputs vehicle is not compatible in general with a process which inputs car (e.g., a bicycle concept is a vehicle, yet, it does not match the car concept).

Another important difference with [66, 41] is the role of the inputs. Our approach assumes that a query exactly specifies the inputs and the outputs of the service to be found, while [66, 41] (possibly) ignore not relevant inputs. For example, in order to search for a book-selling service with the discovery approach of this thesis, a client should formulate a query specifying title, author, deliveryAddress and creditCard as input, and book as output, while a query specifying title and author as input, and book as output would be sufficient for [66, 41]. This is why the approach of this thesis has been designed to (also) address behavioural queries which, in general, will be generated by software developers suitably adapting descriptions of existing services. For example, a software developer that wants to replace a non-working (sub)service of an application can simply use as behavioural query the description of the service to be substituted. It is hence reasonable assuming that such types of queries exactly
3.3. DISCUSSION

specify the requested outputs as well as the provided inputs. Yet, this suggests an interesting line of future work, indeed, the discovery technique presented in this thesis can be suitably extended in order to consider additional inputs. Intuitively, with respect to the functional analysis, the discovery algorithm in Figure 3.13 may be tailored in order to suggest the additional inputs necessary to satisfy the given query, in the style of [27].

A major feature of the functional analysis is surely its ability of finding service compositions. When no single service can satisfy the client request, the functional analysis searches for sets of services capable of satisfying the request, while the single service discovery approaches in [66, 41, 7, 1, 44, 65] simply fail. We also note that service classification-based approaches such as, e.g., [44], are not suitable to feature a composition-oriented discovery, that usually satisfies a query by composing services of different domains: for instance, a travel agency may consist of services belonging to different categories such as transports and economy.

Furthermore, while [66, 7, 65, 27] assumes a global ontology for describing services, the functional analysis crosses ontologies (by suitably interacting with the SemFiT service [39]), since different services usually employ different ontologies. Finally, conversely to [5, 36, 57, 8, 27], the functional analysis guarantees that each determined set of services satisfying the client request is minimal, namely, it contains services strictly necessary to satisfy the request.

With respect to our previous works, the discovery technique presented in this Chapter enhances [27, 21] in crossing different ontologies, while advances [27] in generating minimal compositions of services. Furthermore, the functional analysis works properly also with the critical scenarios of [27], which involve exchanging data across multiple instances of the same service. For example, consider a service implementing a choice between two atomic operations, the first taking as input $x$ and producing as output $y$, and the second generating as output $x$. The choice service is synthetised by two profiles, viz., $(\{x\}, \{y\}, p_1)$ and $(\{\}, \{x\}, p_2)$. A client request asking for $y$ can be properly solved by the set of services $\{p_1, p_2\}$. It is worth mentioning that data and control flow of the composition of $p_1$ and $p_2$ will be handled by the behavioural analysis.

Finally, it is worth observing that, as we will better argue in Chapter 5, we feature a modular organization of the discovery framework, so that independent functionalities (e.g., service registration, discovery, crossing of ontologies) are wrapped into (interacting) distinct modules. Hence, we can easily substitute a module with another one, which, for example, better implements some functionality. In particular, this is the case of the crossing ontology functionality, now implemented by the SemFiT service, that, however, will be substituted, if in the next future more performant ontology crossers will be available.
Chapter 4

Behavioural analysis

In Chapter 3 we described how the (first phase of the) discovery technique introduced in this thesis exploits functional information (viz., functional dependencies within and among services) in order to determine those (sets of) services which satisfy the functional requirements of a given client query. Namely, according to Definition 3.7, a set of services discovered by the functional analysis may be able to return all the required query outputs by taking as input (a sub-set of) the query inputs. Yet, as discussed in the Introduction, in order to guarantee behavioural properties of such sets of services it is mandatory to analyse service behaviour.

In this Chapter we describe the second phase of our discovery technique, that is, the so-called behavioural analysis. The behavioural analysis models service behaviour by means of Open Consume-Produce-Read nets (OCPR for short), a variant of standard Petri nets that we introduced to suitably model some specific assumptions of our discovery framework such as, for example, the abstraction from data multiplicity in modelling services. The OCPR net representation of a service is generated when it enters the local service registry.

Given a set of services – selected, for example, by the functional analysis in Chapter 3 – the behavioural analysis firstly composes them together. Then, it provides two different analyses, according to the type of the client query to be satisfied:

(1) **Lock analysis** – which is performed when a functional query is received. A functional query specifies the (ontology-annotated) inputs and outputs of the desired service, thus not fixing any behavioural constraints. Then, the lock analysis checks whether a (composition of) service(s) satisfies the functional query without locking, that is, whether such a service (given the query inputs) terminates correctly after producing all the requested outputs.

(2) **Bisimilarity analysis** – which is performed when a behavioural query is received. A behavioural query consists of functional requirements, viz., (ontology-annotated) inputs and outputs of the desired service, and behavioural requirements,
vz., a specification of the interaction protocol of the desired service. For instance, a behavioural query can be expressed as the OWL-S process model (see Subsection 2.1.1) of the service to be found, which can be then separately translated by the discovery framework into functional (viz., a list of ontology-annotated inputs and outputs) and behavioural requirements (viz., an OCPR net). The bisimilarity analysis checks whether a (composition of) service(s) features the same (externally observable) behaviour specified by the behavioural query. In this setting, a suitable notion of behavioural equivalence for Web services expressed as OCPR nets is defined.

The rest of the Chapter is organised as follows. Section 4.1 is devoted to describe how the behavioural analysis models service behaviour. In particular, we introduce Consume-Produce-Read nets (CPR nets for short) in Subsection 4.1.1 and we show a first informal encoding from OWL-S descriptions to CPR nets in Subsection 4.1.2. Next, we extend CPR nets in Subsection 4.1.3, where the Open Consume-Produce-Read nets (OCPR nets for short) are defined. Subsection 4.1.4 presents a formal and compositional encoding from OWL-S to OCPR nets, and an example of such an encoding is illustrated in Subsection 4.1.5. Section 4.2 describes how to generate composite services: We discuss how to compose OCPR nets in Subsection 4.2.1, while an example is shown in Subsection 4.2.2. The analysis of the behavioural properties of services is described in Section 4.3. First, a suitable notion of behavioural congruence for Web services is defined in Subsection 4.3.1. Then, Subsection 4.3.2 illustrates how the behavioural analysis verifies the correct termination of a composite service, and how it checks the equivalence between the behaviour of a composite service and the behavioural requirements of a query. An example is presented in Subsection 4.3.3. Finally, related work on nets and net equivalences is discussed in Section 4.4, while some concluding remarks are drawn in Section 4.5.

Preliminary versions of the results presented in this Chapter have been recently published in [16, 18, 17].

4.1 Modelling service behaviour with Petri nets

This Section introduces our chosen formalism to model service behaviour. As anticipated in Section 1.3.3, we employ Petri nets to suitably describe the complete behaviour of a service. We introduce (Subsection 4.1.1) a simple variant of standard condition/event Petri nets (viz., CPR nets for Consume-Produce-Read nets) to naturally model the behaviour of Web services, and in particular the persistence of data. This specific feature of CPR nets is motivated by the fact that we abstract from the multiplicity of data in modelling services.

Intuitively, the mapping from (OWL-S) services to CPR nets presented in the following subsections takes into account the fact that an OWL-S atomic operation can be executed only if all its inputs are available and all the operations that must be
executed before it have been completed. Hence, atomic operations are mapped into transitions, and places and transition firing rules are employed to model both the availability of data (i.e., the data flow) and the executability of atomic operations (i.e., the control flow). Indeed, an atomic operation $T$ is modelled as a transition $t$ having an input/output data place for each input/output of $T$, an input control place to denote that $t$ is executable and an output control place to denote that $t$ has completed its execution.

Given a net $N$ modelling a Web service, we first of all note that the portion of $N$ restricted to transitions and control places should behave as a classical condition/event (C/E) net [76], that is, at most one token should occur for each place. Furthermore, we opted for a model stressing the persistence of data, meaning that, once a data has been produced by some service operation, it has to be kept available for all the service operations that input it. In other words, whilst tokens within control places can be produced and consumed, tokens within data places can be read, produced but not consumed. Hence, the portion of $N$ restricted to data places behaves as a contextual net [58].

To encompass into the same structure the different net behaviour determined by data and control places, we introduce our own particular flavour of contextual C/E nets, which is going to be formally introduced in the following subsection.

We will later discuss in Section 4.4 a few alternative proposals for observational equivalence of Web services, often specified using WS-BPEL, encoded into suitable Petri nets. It is worth mentioning here that a Petri nets semantics\(^1\) for DAML-S (the predecessor of OWL-S) was originally defined by Narayanan and McIlraith in [59]. The proposed mapping was incomplete, since the encoding of the any-order construct was not provided, and most importantly, though, it exploited P/T nets, whose behaviour is quite different from our C/E based proposal. It is also worth mentioning the $E$-service nets [53], namely, nets equipped – similarly to our nets – with control places and input/output message places. Additional orchestration places are employed to compose different $E$-service nets. However, as for [59], also the proposal in [53] is not suitable to represent data persistency and, moreover, neither of them provide any (formal) definition of net composition.

### 4.1.1 CPR nets: Consume-Produce-Read nets

This Subsection introduces consume-produce-read nets. These are a slight extension of standard Petri nets, since they are equipped with two disjoint sets of places, namely, the control (consume-produce) places and the data (produce-read) places.

**Definition 4.1 (CPR nets).** A consume-produce-read net (simply, CPR net) $N$ is a tuple $(C_N, D_N, T_N, F_N, G_N)$ where

\(^1\)Although there are alternative semantics for DAML-S available – e.g., a concurrent execution semantics of DAML-S, based on the operational semantics for concurrent Haskell programs, was introduced in [3] – we consider here Petri net semantics for DAML-S.
- \( C_N \) is a finite set of control places,
- \( D_N \) is a finite set of data places (disjoint from \( C_N \)),
- \( T_N \) is a finite set of transitions,
- \( F_N \subseteq (C_N \times T_N) \cup (T_N \times C_N) \) is the control flow relation,
- \( G_N \subseteq (D_N \times T_N) \cup (T_N \times D_N) \) is the data flow relation.

Our nets behave as standard C/E nets with respect to control places, while data places – once they are inhabited – are never emptied.

As for standard nets [76], we associate a pre-set and a post-set with each transition \( t \), together with two additional sets, called read- and produce-set.

**Definition 4.2 (pre-, post-, read-, and produce-set).** Given a CPR net \( N \), we define for each \( t \in T_N \) the sets

\[
\begin{align*}
\diamondsuit t &= \{ s \in C_N \mid (s, t) \in F_N \} \\
\oslash t &= \{ s \in C_N \mid (t, s) \in F_N \} \\
\blacksquare t &= \{ s \in D_N \mid (s, t) \in G_N \} \\
\blacklozenge t &= \{ s \in D_N \mid (t, s) \in G_N \}
\end{align*}
\]

which denote respectively the pre-set, post-set, read-set and produce-set of \( t \).

Figure 4.1 depicts our chosen graphical notation. Diamonds represent control places, while circles and rectangles represent data places and transitions, respectively. For instance, the transition shown in Figure 4.1 reads the data places labelled \( I_1, I_2, \ldots, I_n \) (this is represented by straight lines) and produces the data places labelled \( O_1, \ldots, O_m \) (this is represented by pointed arrows). In doing so, the control flow passes from the left-most to the right-most control place.

**Definition 4.3 (marking).** Given a CPR net \( N \), a marking \( M \) for \( N \) is a finite set of places in \( P_N = C_N \cup D_N \).
A marking of the net \( N \) coincides with a subset of its set \( P_N \) of places, since each place can contain at most one token. The evolution of a net is given by a relation over markings. A transition \( t \) is enabled by a marking \( M \) if the control places which belong to the pre-set of \( t \) as well as the data places which belong to the read-set of \( t \) are contained in \( M \), and no overlap (as defined below) between \( M \) and the post-set of \( t \) occurs. In this case a firing step may take place. In particular, \( t \) removes the tokens from the control places which belong to the pre-set of \( t \) and adds a token to each place which belongs to the post- and produce-set of \( t \).

Definition 4.4 (firing step). Let \( N \) be a CPR net. Given a transition \( t \in T_N \) and a marking \( M \) for \( N \), a firing step is a triple \( M[\langle t \rangle M'] \) such that \((\diamond t \cup \bullet t) \subseteq M \) and \((M \cap \diamond t) \subseteq \diamond t \) (\( M \) enables \( t \)), and moreover \( M' = (M \setminus \diamond t) \cup t \diamond t \cup \bullet t \).

We write \( M[\langle t \rangle M'] \) if there exists some \( t \) such that \( M[\langle t \rangle M'] \).

The enabling condition states that (1) all the tokens of the pre-set of a transition have to be contained in the marking, and (2) that the marking does not contain any token in the post-set of the transition, unless it is consumed and regenerated. The second condition usually characterizes C/E nets.

Note instead that data places act as sinks, hence, any token can be added and only the occurrence of a token is checked. The read-only feature of data places is reminiscent of the work on so-called contextual C/E nets by Montanari and Rossi [58].

4.1.2 From OWL-S descriptions to Petri nets

In this Subsection we informally discuss how OWL-S service descriptions can be mapped into CPR nets. For the sake of clarity, a formal presentation of the encoding is delayed until Subsection 4.1.4, after the introduction of open CPR nets and of their semantics.

Let us define a service (i.e., a composite process) as a triple \((i, P, f)\) where \( P \) denotes the CPR (sub)net representing the service and \( i \) and \( f \) denote the initial and the final control places of \( P \), respectively. Intuitively, to define compositional operators it is sufficient to properly coordinate the initial and final control places of the employed services. For instance, let us consider the sequential composition of two services \((i_1, P_1, f_1)\) and \((i_2, P_2, f_2)\). This is a CPR net consisting of the two services and a transition whose initial control place is \( f_1 \) and the final control place is \( i_2 \). By doing so, \( P_1 \) has to be completed before \( P_2 \) can start.

We do not give here a formal semantics of the sequence operator and of the other OWL-S control constructs. Anyway, Figure 4.2 provides an intuitive illustration of how OWL-S composite operations can be mapped into CPR nets. The \( PX \)-labelled boxes represent \((i_x, PX, f_x)\) services, the dark grey rectangles identify empty transitions, and the light grey diamonds denote the initial and final control places of the resulting nets. To simplify reading we omitted data places from the nets of Figure 4.2. Yet, it is important to note that the \( PX \)-boxes can share data places.
to simulate the exchange of data amongst services, as we will formally define in Subsection 4.1.4.

4.1.3 OCPR nets: Open Consume- Produce-Read nets

A first step for defining compositionality is to equip nets with a notion of interface and context. Roughly speaking, an interface represents the “ports” which can be used to interact with a net, while a context is an external environment where a net can be embedded in. Intuitively, the net and the context will interact through the net interface. For the sake of presentation, a chosen net N = (C_N, D_N, T_N, F_N, G_N) is assumed.

**Definition 4.5 (Open CPR net).** Let N be a CPR net. An interface for N is a triple \( \langle i, f, OD \rangle \) such that \( i \neq f \) and

- \( i \) is a control place (i.e., \( i \in C_N \)), viz., the initial control place,
- \( f \) is a control place (i.e., \( f \in C_N \)), viz., the final control place, and
- \( OD \) is a set of data places (i.e., \( OD \subseteq D_N \)), viz., the open data places.

An interface is an outer interface O for N if there exists no transition \( t \in T_N \) such that either \( i \in t^\ominus \) or \( f \in t^\oplus \).

An open CPR net \( \mathcal{N} \) (OCPR for short) is a pair \( \langle N, O \rangle \), for N a CPR net and O an outer interface for N.

We symmetrically define an inner interface for N as an interface such that there is no transition \( t \in T_N \) verifying either \( f \in t^\ominus \) or \( i \in t^\oplus \).
4.1. MODELLING SERVICE BEHAVIOUR WITH PETRI NETS

The condition characterising outer interfaces requires that the initial control place has no incoming transition and the final control place has no outgoing transition. Hence, OCPR nets recall the WorkFlow nets proposed by van der Aalst [86]², as later discussed in Section 4.4.

The components of a specific interface or open net are often denoted by adding them a subscript. For instance, the triple \( \langle i_I, f_I, OD_I \rangle \) denotes the initial control place, the final control place and the open data places, respectively, of the interface \( I \).

Given an open net \( N \), \( Op(N) \) denotes the set of open places, which consists of those places occurring on the outer interface, initial and final places included. Furthermore, the places of \( N \) not belonging to \( Op(N) \) constitute the closed places. It is important to note that open places are crucial for the definition of the observational equivalence that we propose in Subsection 4.3.1.

Figure 4.3 shows the graphical notation used to represent OCPR nets. The bounding box of the OCPR net \( WS_1 \) represents the outer interface of the net. Note the initial and final places used to compose the control of services (as suggested in Figure 4.2) as well as the open data places employed to share data.

![Graphical notation for OCPR nets](image)

**Figure 4.3: Two open nets, a context and a composite net.**

**Definition 4.6 (CPR context).** A CPR context \( C[-] \) is a triple \( \langle N, O, I \rangle \) such that \( N \) is a CPR net, \( I = \langle i_I, f_I, OD_I \rangle \) and \( O = \langle i_O, f_O, OD_O \rangle \) are an inner and an outer interface for \( N \), respectively, and \( i_I \neq f_O \), \( i_O \neq f_I \).

An example of context \( C[-] \) is shown in Figure 4.3. Basically, it is an open net with an hole, represented there by a grey area. The border of the hole denotes

---

²More precisely, our nets lack the connectiveness requirement, see e.g. [86, Section 2.2]. Note however that, even if this is not made explicit, the property holds for all the nets obtained by modelling OWL-S composite operations.
the inner interface of the context. As for open nets, the bounding box is the outer
interface. The only difference between inner and outer interfaces is that the initial
place of the former has no outgoing transitions (and vice versa for final places).

Contexts represent environments in which services can be embedded, i.e., possible
ways they can be used by other services. An OCPR net can be inserted in a context
if the net outer interface and the context inner interface must coincide.

**Definition 4.7 (OCPR composition).** Let \( N = \langle N, O \rangle \) be an OCPR net and
\( C[-] = \langle N_{C}, O_{C}, I_{C} \rangle \) a CPR context, such that \( O = I_{C} \). Then, the composite net
\( C[N] \) is the OCPR net \( (C_{N} \uplus O_{N_{C}}, D_{N} \uplus D_{N_{C}}, T_{N} \uplus T_{N_{C}}, F_{N} \uplus F_{N_{C}}, I_{N} \uplus I_{N_{C}}) \) with
outer interface \( O_{C} \).

In other words, the disjoint union of the two nets is performed, except for those
places occurring in \( O \), which are coalesced: this is denoted by the symbol \( \uplus \).
Moreover, \( O_{C} \) becomes the set of open places of the resulting net.

More precisely, let \( p_{I} \) and \( p_{O} \) denote two places occurring in the inner interface
\( I_{C} \) of the context \( C[-] \) and in the outer interface \( O \) of the net \( N \), respectively. If
\( p_{I} \) is equivalent to \( p_{O} \), they are collapsed in a single place \( p \) in the composite net
\( C[N] \). It is worth noting that \( p_{I} \) is considered equivalent to \( p_{O} \), if the data (viz., an
ontology concept) represented by \( p_{I} \) is a subConcept (according to Definition 3.4 of
Subsection 3.1.1) of the data represented by \( p_{O} \).

Consider, for instance, the net \( WS_{1} \), the context \( C[-] \) and their composition,
denoted by \( C[WS_{1}] \), as illustrated in Figure 4.3. The places on the outer interface
of \( WS_{1} \) are coalesced with the ones on the inner interface of \( C[-] \). The output
interface of \( C[WS_{1}] \) is the outer interface of \( C[-] \). Note that the data place \( A \) is
open in \( WS_{1} \) and closed in \( C[WS_{1}] \): this example highlights the capability of hiding
places, removing them from the outer interface of an open net. Indeed, this feature
is reminiscent of the restriction operator of process calculi, such as CCS [55].

### 4.1.4 A compositional encoding for OWL-S

We now define a formal OCPR encoding for OWL-S service descriptions. To this
aim, we first introduce the notion of binary contexts, and then we use it for modelling
composite services.

As depicted in Figure 4.1, an atomic process is encoded in a single transition
net. Instead, the encoding of a composite service requires to extend the notion of
interface, in order to accommodate the plugging of two nets into a context.

**Definition 4.8 (binary contexts).** A CPR binary context \( C[-,-] \) is a tuple
\( \langle N, O, I_{1}, I_{2} \rangle \) such that the triples \( \langle N, O, I_{1} \rangle \), \( \langle N, O, I_{2} \rangle \) are CPR contexts, and
\( \{i_{1}, f_{1}\} \cap \{i_{2}, f_{2}\} = \emptyset \).

Since it suffices for our purposes, we restrict our attention to binary contexts, the
general definition being easily retrieved. Note that the control places of the inner
interfaces are all different, while no condition is required for data places.
Definition 4.9 (binary composition). Let $N_1 = \langle N_1, O_1 \rangle$, $N_2 = \langle N_2, O_2 \rangle$ be OCPR nets, and $C[-1, -2]$ a CPR binary context, such that $O_1 = I_1$ and $O_2 = I_2$. Then, the composite net $C[N_1, N_2]$ is the OCPR net $(C_{N_1} \uplus U C_{N_2} \uplus U C_N, D_{N_1} \uplus U D_{N_2} \uplus U D_N, T_{N_1} \uplus \uplus T_{N_2} \uplus T_N, F_{N_1} \uplus \uplus F_{N_2} \uplus \uplus F_N, G_{N_1} \uplus \uplus G_{N_2} \uplus \uplus G_N)$ with outer interface $O_C$.

As for unary contexts, the disjoint union of the three nets is performed, except for coalescing those places occurring either in $O_1$ or $O_2$ (denoted by $\uplus U$).

Next, we define the extension of a context for a set of data places.

Definition 4.10 (context extension). Let $A$ be a set of data places, $C[-] = \langle N, O, I \rangle$ be a context. The context extension $C_A[-]$ is the context with net $N_A = (C_N \uplus U A, T_N, F_N, G_N)$, inner interface is $I_A = \langle i_I, f_I, OD_I \cup A \rangle$ and outer interface is $O_A = \langle i_O, f_O, OD_O \cup A \rangle$.

Data places (i.e., $D_N \uplus U A$) are obtained by disjoint union, except for coalescing those places occurring either in $O$ or $I$ (denoted by $\uplus U$). An analogous operation is defined for binary contexts. As an example, consider the context $choice_A[-1, -2]$, illustrated in the right side of Figure 4.4, for $A = \{A_1, \ldots, A_n\}$. It is the extension of the $choice[-1, -2]$ context (i.e., the left side of Figure 4.4) that just contains four transitions and six control places.

![Figure 4.4: Context and extended context of the choice operator.](image)

In order to define the encoding, for each OWL-S operator $op$ we define a corresponding (possibly binary) context $op[-]$. Figure 4.5 illustrates the encoding for the (other) OWL-S operators, i.e., sequence, any-order, split, split+join, repeat-until and repeat-while. Note that the contexts depicted in Figure 4.5 are the extensions $op_A[-]$ of contexts $op[-]$ corresponding to $op$, for $A = \{A_1, \ldots, A_n\}$.

Now we can give the formal encoding. Let $S$ be an OWL-S process model and let $A$ be a set of data places, containing all the data occurring in $S$. The encoding of $S$ with $A$ open places is inductively defined as follows:
CHAPTER 4. BEHAVIOURAL ANALYSIS

Figure 4.5: Contexts of the OWL-S operators extended for $A = \{A_1, \ldots A_n\}$.

$\| S \|_A = \begin{cases} 
N_{S,A} & \text{if } S \text{ is atomic,} \\
op_A[\| S_1 \|_A] & \text{if } S = \text{op}(S_1), \\
op_A[\| S_1 \|_A, \| S_2 \|_A] & \text{if } S = \text{op}(S_1, S_2), 
\end{cases}$

where $N_{S,A}$ is the OCPR net with a single transition that reads all the input data of the atomic service $S$, and produces all the output data of $S$ (as illustrated by Figure 4.1), extended with all the data places of $A$.

For the sake of simplicity, we left implicit the possible renaming of control places, needed for the composition of nets and contexts to be well-defined.

It is also worth noting that the encoding of conditional execution, viz., if-then-else, coincides with the encoding of non-deterministic execution, viz., choice. Indeed, our implementation of the process model abstracts away from data values.
4.1. MODELLING SERVICE BEHAVIOUR WITH PETRI NETS

4.1.5 An example

Let us consider the sample services HotelService and RatingOne illustrated in Figures 2.6 and 2.4 of Subsection 2.1.2. In this subsection, we describe how to map their process models into OCPR nets according to the encoding defined in Subsection 4.1.4.

Let us start by translating into OCPR nets the OWL-S process model of the HotelService (Figure 2.6). HotelService has three atomic processes (i.e., selectHotel, payment, collectInfoHotel). Each atomic process \( p \) corresponds to an OCPR net with a single transition that reads all the input data of \( p \) and produces all the output data of \( p \). For example, consider the selectHotel atomic process of HotelService. The corresponding OCPR net is depicted in the left side of Figure 4.6. Note that the single transition of such a net has an initial control place \( i \), a final control place \( f \), three input data places (i.e., hotel, beginDate, lastDate) and an output data place (i.e., accommodationFee), all belonging to the net interface.

Next, we perform the sequential composition of the atomic processes selectHotel and payment. According to Definition 4.9, we perform the disjoint union of the nets modelling the atomic processes selectHotel and payment (see Figure 4.6), and the sequence \( A[-1,-2] \) context (see Figure 4.5), where \( A = \{ \text{hotel, beginDate, lastDate,} \)
creditCard, contactInformation, accommodationFee, hotelInvoice}. The resulting net is illustrated in Figure 4.7.

![Figure 4.8: OCPR net representation of the HotelService (Step 2).](image)

Then, we model the non-deterministic choice between the collectInfoHotel atomic process and the sequential execution of the selectHotel and payment processes. Hence, we perform the disjoint union of the net depicted in Figure 4.7, and of the net modelling the choice\(_A[-1, -2]\) context (see Figure 4.4), where \(A = \{\text{hotel, beginDate, lastDate, creditCard, contactInformation, state, city, accommodationFee, hotelInvoice, infoHotel}\}\). The resulting net is shown in Figure 4.8.

![Figure 4.9: OCPR net representation of the HotelService.](image)

Finally, in order to complete the translation from OWL-S to OCPR net of HotelService, we have to model the iterative execution of the sub-tree of the HotelService
process model (Figure 2.6) with root choice #2. Hence, we perform the disjoint union of the net depicted in Figure 4.8, and of the net modelling the repeat-until \([\mathcal{A}[-1, -2]]\) context (Figure 4.5), where \(\mathcal{A} = \{\text{hotel}, \text{beginDate}, \text{lastDate}, \text{creditCard}, \text{contactInformation}, \text{state}, \text{city}, \text{accomodationFee}, \text{hotelInvoice}, \text{infoHotel}\}\). Figure 4.9 shows the complete OCPR net representation of the HotelService.

Figure 4.10: OCPR net representation of the RatingOne service (Step 1).

Let us now consider the RatingOne service. As one can observe in Figure 2.4 of Subsection 2.1.2, the RatingOne process model consists of a sequence of a split+join composite process and of the averageRating atomic process. Let us start by considering the split+join process. Since it consists of three atomic processes (i.e., computeFirstRating, computeSecondRating and computeThirdRating), we first perform the parallel composition of computeSecondRating and computeThirdRating. To do that, we employ the split+join \([\mathcal{A}[-1, -2]]\) context (Figure 4.5), with \(\mathcal{A} = \{\text{amountOfCredit, balance, guarantee, secondRating and thirdRating}\}\). The OCPR net modelling the parallel composition of computeSecondRating and computeThirdRating is depicted in Figure 4.10.

Figure 4.11: OCPR net representation of the RatingOne service (Step 2).
We next perform the parallel composition of the net of Figure 4.10 and of the net modelling the computeFirstRating atomic process (not depicted here). We employ again the split+join,\( A[−1,−2]\) context, with \(A = \{\text{amountOfCredit}, \text{balance}, \text{guarantee}, \text{firstRating}, \text{secondRating}, \text{thirdRating}\}\). The OCPR net modelling the split+join process of the RatingOne service is illustrated in Figure 4.11.

![Figure 4.11: OCPR net representation of the RatingOne service.](image)

Finally, we employ the sequence,\( A[−1,−2]\) context (Figure 4.5), with \(A = \{\text{amountOfCredit}, \text{balance}, \text{guarantee}, \text{firstRating}, \text{secondRating}, \text{thirdRating}, \text{rating}\}\), to perform the sequential execution of the split+join composite process and the averageRating atomic process of RatingOne. The OCPR net properly representing the RatingOne service is shown in Figure 4.12.

![Figure 4.12: OCPR net representation of the RatingOne service (Step 3).](image)

### 4.2 From sets of services to compositions of services

It is worth stressing here that, so far, we have been intentionally using the expression “sets of services” instead of “compositions of services”, since the functional analysis does not guarantee any behavioural properties of the discovered services. Hence, for example, a set of services which satisfies the functional requirements of the query may deadlock. Verifying whether such services can really be composed together without dead-locking is the first objective of the behavioural analysis.

Given a set of services returned by the functional analysis, the behavioural analysis checks whether such services can properly interact, so as to really satisfy the query (functional) requirements. In particular, the behavioural analysis:

1. considers the OCPR net representations of the services in the set, and builds the parallel composition of such nets, and next
(2) analyses the resulting OCPR net in order to verify whether it produces the query outputs starting from the query inputs without dead-locking.

We describe step (1) in Subsection 4.2.1, while we delay the description of step (2) until Subsection 4.3.2, after the introduction of the labelled transition system (Definition 4.17) distilled from the firing semantics of OCPR nets.

### 4.2.1 Constructing the composite service

Let \( S = \{s^1_1, s^2_1, ..., s^k_1, s^1_2, ..., s^k_2, ..., s^1_n, ..., s^m_n\} \) be a set of services returned by the functional analysis. More precisely, \( S \) is a set of service profiles. We write \( s^j_i \) to denote the profile \( j \) of the service \( s_i \). For each profile \( s^j_i \in S \), the behavioural analysis considers the net \( \mathcal{N}_{s_i} \), namely, the OCPR net representation\(^3\) of (the complete behaviour of) the service \( s_i \). Yet, \( s^j_i \) denotes a particular profile of \( s_i \), which may partially employ the inputs and outputs of the whole service \( s_i \). The encoding presented in Subsection 4.1.4 maps an OWL-S service into an OCPR net, where all the data places are open, i.e., they belong to the outer interface. Let \( O_{\mathcal{N}_{s_i}} \) denotes the outer interface of \( \mathcal{N}_{s_i} \), then, the behavioural analysis removes from \( O_{\mathcal{N}_{s_i}} \), those data places which do not correspond to the inputs and outputs of the profile \( s^j_i \). In other words, the behavioural analysis closes those data places which do not need to be observed.

**Definition 4.11 (hiding context).** Let \( O = \langle i_O, f_O, OD_O \rangle \) be an outer interface and \( A \) a set of data places such that \( A \subseteq OD_O \). The hiding context (with respect to \( A \) and \( O \)) \( \nu_{A,O}[-] \) is the context whose net has no transition, whose inner interface is \( O \) and whose outer interface is \( \langle i_O, f_O, OD_O \setminus A \rangle \).

Now, let \( A_{ij} \) denote the set of data places of \( \mathcal{N}_{s_i} \) that do not correspond to the inputs and outputs of the profile \( j \) of \( s_i \). Given a profile \( s^j_i \), the behavioural analysis considers the OCPR net \( \mathcal{N}_{ij} = \nu_{A_{ij}, O_{\mathcal{N}_{s_i}}} [\mathcal{N}_{s_i}] \).

Summing up, given a set of profiles \( S = \{s^1_1, s^2_1, ..., s^k_1, s^1_2, ..., s^k_2, ..., s^1_n, ..., s^m_n\} \), the behavioural analysis performs the parallel composition of the nets \( \mathcal{N}_{11}, \mathcal{N}_{12}, ..., \mathcal{N}_{1h}, \mathcal{N}_{21}, ..., \mathcal{N}_{2k}, ..., \mathcal{N}_{n1}, \mathcal{N}_{n2}, ..., \mathcal{N}_{nm} \).

It is worth noting that, if the set of profiles returned by the functional analysis contains \( n \) profiles of the same service \( s \), we insert into the composite net \( n \) copies of the OCPR net modelling \( s \), each of them possibly providing a different interface. In other words, we are considering multiple executions of the same service. For example, let us consider a service implementing a choice between two atomic operations, the first taking as input \( A \) and producing \( B \), and the second taking as input \( C \) and producing \( D \). A query asking for a service that inputs \( A \) and \( C \), and produces \( B \) and \( D \) can be solved by executing the choice service twice. This is modelled by

\(^3\)As described in the Introduction (Subsection 1.3.3), the OCPR net representation of a service \( s \) is generated when \( s \) enters the local service repository of the discovery framework.
parallel composition of two copies of the OCPR net representing the choice service, exposing the interfaces \( \{ A, B \} \), and \( \{ C, D \} \), respectively.

Given two OCPR nets \( N_{ij}^1 = \langle N_{ij}^1, O_{ij}^1 \rangle \) and \( N_{ij}^2 = \langle N_{ij}^2, O_{ij}^2 \rangle \), the behavioural analysis computes their parallel composition by performing the disjoint union of \( N_{ij}^1, N_{ij}^2 \) and of the context \( \text{split} + \text{join}(O_{ij}^1, O_{ij}^2)[-1, -2] \), except for coalescing those places occurring either in \( O_{ij}^1 \) or \( O_{ij}^2 \), according to Definitions 4.9,4.10 of Subsection 4.1.4. More precisely, let \( o_1, o_2 \) be two (open) data places occurring in \( O_{ij}^1 \) and \( O_{ij}^2 \), respectively, and let \( c_1, c_2 \) the ontology concepts represented by \( o_1 \) and \( o_2 \). Then, \( o_1 \) and \( o_2 \) are coalesced into the single (open) data place \( o_1 \) if \( c_1 \) is syntactically and/or semantically equivalent to \( c_2 \) (i.e., \( c_1 \) is a \textit{subConcept} of \( c_2 \), as per Definition 3.4).

Note that such semantic relationships may be suggested by the functional analysis, that determined them when executing the discovery algorithm in Subsection 3.2.1.

Finally, it is worth noting that the resulting interface of the composite net may contain some places which, yet, it is not necessary to observe, since, e.g., they do not belong to the client request. Then, such places are removed from the interface of the composite net.

### 4.2.2 An example

Let us now continue the example previously introduced in Subsection 3.2.4. We consider first the query asking for a service which inputs \textit{internationalConference}, \textit{contactInformation}, \textit{creditCard} and \textit{hotel}, and yields \textit{registrationReceipt} and \textit{invoice}. As discussed in Subsection 3.2.4, the functional analysis returns two sets of services capable of satisfying such a query. Let us consider the first set, which includes the profile \( \{ \text{hotel, contactInformation, creditCard, beginDate, lastDate} \} \) and \( \{ \text{accomodationFee, hotelInvoice} \} \), \( H_2 \) of \textit{HotelService} and the profile \( \{ \text{academicEvent, contactInformation, creditCard} \} \), \{\textit{registrationFee, city, country, startDate, endDate, registrationReceipt}, \( C_1 \) of \textit{ConferenceService}.

Let \( N_H \) and \( N_C \) be the OCPR nets modelling the behaviour of \textit{HotelService} (Figure 4.9) and \textit{ConferenceService} (Figure 4.13), respectively. As one can observe in Figure 4.9, the interface of \( N_H \) contains the data places \textit{hotel, beginDate, lastDate, state, city, creditCard, contactInformation, infoHotel, accomodationFee and hotelInvoice}, while profile \( H_2 \) contains only \textit{hotel, contactInformation, creditCard, beginDate, lastDate, accomodationFee and hotelInvoice}. Hence, the behavioural analysis removes city, state and infoHotel from the interface of \( N_H \). Let us now consider the net \( N_C \).

As one can note in Figure 4.13, the interface of \( N_C \) contains a data place, viz., \textit{bankAccount} which does not belong to the profile \( C_1 \) of \textit{ConferenceService}. Hence, the behavioural analysis removes it from the interface of \( N_C \).

Next, the parallel composition of \( N_H \) and \( N_C \) is performed. More precisely, the behavioural analysis composes the \textit{split}+\textit{join} context (see Figure 4.5) with the nets \( \nu_{A_H, O_H}[N_H] \) and \( \nu_{A_C, O_C}[N_C] \), where \( A_H = \{ \text{city, state, infoHotel} \} \), \( A_C = \{ \text{bankAccount} \} \), and \( O_H, O_C \) denote the interfaces of \( N_H \) and \( N_C \), respectively. The
4.2. FROM SETS OF SERVICES TO COMPOSITIONS OF SERVICES

resulting composite net is shown in Figure 4.14. Indeed, note that the interface of such a net does not include the data places city, state, infoHotel and bankAccount. Furthermore, it is also worth noting that the data places beginDate and lastDate on the interface of $N_H$ and the data places startDate and endDate on the interface of $N_C$ have been collapsed into places startDate and endDate on the interface of the resulting composite net. Indeed, as discussed in the example of Subsection 3.2.4, the concepts beginDate and startDate, even if syntactically different, are semantically equivalent, as well as the concepts lastDate and endDate, which feature the same semantic meaning.

Let us now consider the second set of services returned by the functional analysis for the first query of Subsection 3.2.4, which includes the profile (\{hotel, contactInformation, creditCard, beginDate, lastDate, city, state\}, \{accomodationFee, hotelInvoice, infoHotel\}, $H_3$) of HotelService and the profile (\{academicEvent, contactInformation, creditCard\}, \{registrationFee, city, country, startDate, endDate, registrationReceipt\}, $C_1$) of ConferenceService. Similarly to the previous example, the behavioural analysis performs the parallel composition of the nets $N_H$ and $\nu_{AC,OC}[N_C]$. In particular, note that the profile $H_3$ of HotelService employs all the data places which occur on the interface of the net $N_H$, so that it is not necessary to apply the hiding context to $N_H$. The net resulting from the parallel composition of $N_H$ and $\nu_{AC,OC}[N_C]$ (not depicted here) is similar to the net of Figure 4.14, except for data places state, city and infoHotel, which are open in the composite net.

Consider now the second sample query introduced in Subsection 3.2.4 asking for a service which inputs amountOfCredit, guarantee and balance, and returns rating. The functional analysis finds four sets of services satisfying that query, viz., \{RO$_1$\}, \{RT$_1$\}, \{RT$_2$\} and \{RT$_3$\}, corresponding to the single profile of RatingOne and to the three profiles of RatingTwo, respectively. Let us consider the first set, which consists of the single profile (\{amountOfCredit, balance, guarantee, firstRating, secondRating,
thirdRating}, \{firstRating, secondRating, thirdRating, rating\}, \{RO_1\}) of the RatingOne service (Figure 2.4). The behavioural analysis simply considers the OCPR net \(N_{RO}\) modelling the behaviour of RatingOne and illustrated in Figure 4.12, not applying the hiding context to \(N_{RO}\), since the profile \(RO_1\) employs all the data places occurring on the interface of \(N_{RO}\).

Consider now the sets of services \{\(RT_1\}\}, \{\(RT_2\}\) and \{\(RT_3\)\} (namely, the three profiles of RatingTwo). For each of them the behavioural analysis directly considers the net \(N_{RT}\) depicted in Figure 4.15, which models the behaviour of the RatingTwo service. Yet, in case of the profile \(RT_1\) (viz., \{\{amountOfCredit, balance, guarantee, firstRating\}, \{firstRating, rating\}, \(RT_1\))\), the behavioural analysis applies the hiding context \(\nu(A_{RT_1},O_{RT})\) to \(N_{RT}\), where \(A_{RT_1} = \{secondRating, thirdRating\}\) and \(O_{RT}\) denotes the interface of \(N_{RT}\). Moreover, in case of the profile \(RT_2\) (viz., \{\{amountOfCredit, balance, guarantee, firstRating, secondRating\}, \{firstRating, secondRating, rating\}, \(RT_2\))\), the behavioural analysis applies the hiding context \(\nu(A_{RT_2},O_{RT})\) to \(N_{RT}\), where \(A_{RT_2} = \{thirdRating\}\).

### 4.3 Reasoning on service behaviour

In this Section we discuss how the behavioural analysis fruitfully exploits service behaviour in order to check:

1. whether the services in a given set \(S\) will interact properly, thus generating the requested outputs without locking, and
4.3. REASONING ON SERVICE BEHAVIOUR

Figure 4.15: OCPR net representation of the RatingTwo service.

(2) whether the services in a given set $S$, suitably composed together, feature the same (externally observable) behaviour of a given client request, where $S$ is a set of services previously determined by the functional analysis. The rest of the Section is organised as follows. In Subsection 4.3.1 we introduce a notion of behavioural equivalence for Web services, while in Subsection 4.3.2 we discuss suitable algorithms for the verification of the termination and of the equivalence of services. Finally, an example is illustrated in Subsection 4.3.3.

4.3.1 A behavioural congruence for Web services

In order to satisfy behavioural queries and determine whether a given service (or service composition) features a desired interaction behaviour, a well-founded notion of behavioural equivalence for services is needed. The objective of this Subsection is to introduce a decidable notion of equivalence between Web services represented as OCPR nets. Such a notion establishes whether two service behaviour are equivalent and it is employed by the behavioural analysis in order to perform the bisimilarity analysis (so to solve behavioural client queries). Yet, the availability of such a notion allows to further address other crucial issues in service-oriented computing, e.g., service replaceability and modular service development. Summing up, the notion of behavioural equivalence introduced in this Section can be employed to suitably address the following issues:

(1) matching services — to check whether a (composition of) service(s) matches a query that specifies the behaviour of the desired service (composition) to be found,
(2) incremental development of services — to check whether two different versions of a service are equivalent,

(3) publication of correct service specifications — to check whether a (complex) service implements a given specification,

(4) replaceability of services — to check whether a service \( s \) which takes part in a composition \( C[s] \) can be replaced with a different service \( r \) without changing the behaviour of \( C \), i.e., guaranteeing the equivalence between \( C[s] \) and \( C[r] \).

We can hence outline the main features that a suitable notion of equivalence should have, that is, weakness and compositionality. It has to be weak as it must equate services with respect to their externally observable behaviour. Indeed, it is reasonable that a simple query can be satisfied by a complex service composition, that two versions of a service differently implement the same operations, as well as that a service specification hides unnecessary and/or confidential details of its implementation. Therefore, this notion of equivalence has to be abstract enough to equate services that differ only on internal transition steps. Furthermore, service replaceability also asks for compositionality, and if two services are equivalent, then they can be always used interchangeably.

Let us now consider some examples (inspired by [48]). Figure 4.16 illustrates the OCPR nets of seven services. Consider the services \( W S_1 \) and \( W S_2 \) of Figure 4.16. As one may note, \( W S_1 \) and \( W S_2 \) have the same behaviour with respect to the notion of trace equivalence. Indeed, they have identical sets of traces, since after producing \( A \) they may alternatively read either \( B \) or \( C \). Consider now the context \( C_1[-] \), depicted in Figure 4.16, which represents a possible environment in which \( W S_1 \) and \( W S_2 \) can be embedded. We note that \( C_1[-] \) inputs \( A \), produced by \( W S_1 \), and yields \( B \) or \( C \), taken as input by \( W S_1 \). Hence, the composition \( C_1[W S_1] \) works and finishes properly. Now, in order to test if the trace equivalence is the notion suitable for our purpose, we replace \( W S_1 \) with the trace equivalent service \( W S_2 \) and we check whether the composition \( C_1[W S_2] \) works properly as well. Yet, \( C_1[W S_2] \) produces a possibly dead-locking system.

Let us now describe a second example arguing for the need of weakness. Consider the services \( W S_3 \) and \( W S_4 \). For instance, \( W S_4 \) could be a composition candidate to satisfy the query represented by \( W S_3 \). Although \( W S_3 \) and \( W S_4 \) appear different as they perform a different number of transitions, they both produce \( B \) or \( C \). Namely, \( W S_3 \) and \( W S_4 \) have identical externally observable behaviour, and they indeed should be considered equivalent.

By taking into account the requirements briefly outlined above, we define a novel notion of equivalence based on bisimilarity which features weakness, compositionality and decidability. In particular, in this Subsection, we introduce two notions of equivalence: the first is conceptually the correct one, even if it turns out to be quite hard to reason about, while the second provides a simple, decidable equivalent characterization of the former.
4.3. REASONING ON SERVICE BEHAVIOUR

A saturated bisimilarity for OCPR nets

This Section addresses the question of the equivalence between nets. Our answer relies on an observational approach, equating two systems if they can not be told apart by an external observer. More precisely, the observer can only examine the open places of a net, which is otherwise a black box, and only those places may be checked for verifying whether they are actually inhabited or are empty.

For the sake of presentation, a chosen OCPR net $\mathcal{N} = (N, O)$ is assumed. We also remind that $Op(N)$ denotes the open places of $\mathcal{N}$.

**Definition 4.12 (observation).** Let $\mathcal{N}$ be an OCPR net, and $M$ a marking of $N$. The observation on $\mathcal{N}$ at $M$ is the set of places $Obs(\mathcal{N}, M) = Op(\mathcal{N}) \cap M$.

Thus, an observer can look the evolution of the system by observing whether tokens are produced or consumed in the open places. Accordingly, two OCPR nets $\mathcal{N}$ and $\mathcal{N}'$ with the same outer interface and with initial markings $M$ and $M'$ are considered equivalent if $Obs(\mathcal{N}, M) = Obs(\mathcal{N}', M')$, namely, the markings $M$ in $\mathcal{N}$ and $M'$ in $\mathcal{N}'$ contain the same open (viz., observable) inhabited places, and if every state reachable from $M$ in $\mathcal{N}$ is equivalent to a state reachable from $M'$ in $\mathcal{N}'$ (and vice versa).
The previous remark is formalized by the definition below, where $\mathcal{MN}$ denotes the set of all OCPR nets with markings and $\rightarrow_{\mathcal{N}}$ denotes the reflexive and transitive closure of the firing relation $\rightarrow$ for the net $N$ underlying $\mathcal{N}$.

**Definition 4.13 (naive bisimulation).** A symmetric relation $R \subseteq \mathcal{MN} \times \mathcal{MN}$ is a naive bisimulation if whenever $(N, M) R (N', M')$ then

- $O_N = O_{N'}$ and $\text{Obs}(N, M) = \text{Obs}(N', M')$, and
- $M \rightarrow_{\mathcal{N}} M_1$ implies $M' \rightarrow_{\mathcal{N}} M'_1 \in R (N, M_1)$.

The union of all naive bisimulations is called naive bisimilarity.

The above equivalence is “naive” in the sense that it clearly fails to be compositional. Indeed, consider the OCPR nets $WS_1$ and $WS_2$ in Figure 4.3. They are trivially equivalent since none of them can fire. However, if we insert them into a context with a transition generating a token in the initial place and in the data place $A$, we obtain two different contexts (one can now produce a token on $B$ reaching the final state $f$, while the latter can not move).

The solution out of the impasse, which is quite standard both in functional languages and process calculi, is to allow the observer to perform more complex experiments, inserting a net into any possible context.

**Definition 4.14 (saturated bisimulation).** A symmetric relation $R \subseteq \mathcal{MN} \times \mathcal{MN}$ is a saturated bisimulation if whenever $(N, M) R (N', M')$ then

- $O_N = O_{N'}$ and
- $\text{Obs}(N, M) = \text{Obs}(N', M')$, and
- $\forall C[-]. M \rightarrow_{C[N]} M_1$ implies $M' \rightarrow_{C[N']} M'_1 \in R (C[N], M_1)$.

The union of all saturated bisimulations is called saturated bisimilarity ($\approx_S$).

Clearly, $\approx_S$ is by definition a congruence. Indeed, it is the largest bisimulation that preserves compositionality, as stated below.

**Proposition 4.1** $\approx_S$ is the largest bisimulation that is also a congruence.

The above proposition ensures the compositionality of the equivalence, hence, the possibility of replacing one service by an equivalent one without changing the behaviour of the whole composite service. Moreover, the equivalence is “weak” in the sense that, differently from most of the current proposals, no explicit occurrence of a transition is observed. It is worth noting that the proofs of propositions and theorems contained in this Section can be found in [18] as well as in the Appendix of this thesis.

The previous definition leads to the following notion of equivalence between OCPR nets, hence, between services.
Definition 4.15 (bisimilar nets). Let $\mathcal{N}, \mathcal{N}'$ be OCPR nets. They are bisimilar, denoted by $\mathcal{N} \approx \mathcal{N}'$, if $(\mathcal{N}, \emptyset) \approx_S (\mathcal{N}', \emptyset)$.

The choice of the empty marking guarantees that the equivalence is as general as possible. Indeed, the presence of a token in an open place can be simulated by closing the net with respect to a transition adding a token in that place, and if any two nets are saturated bisimilar with respect to the empty marking, they are so also with respect to any marking with tokens in the open places.

The negative side of $\approx$ is that this equivalence seems quite hard to be automatically decided because of the quantification over all possible contexts. In the following subsection we introduce an alternative equivalence, easier to reason about and to automatically verify, and we prove that it coincides with $\approx_S$.

An equivalent decidable bisimilarity

Saturated bisimulation seems conceptually the right notion, and this is further argued in Section 4.4. However, it seems quite hard to analyze (or automatically verify), due to the universal quantification over contexts. In this subsection we thus introduce weak bisimilarity, based on a simple labelled transition system (LTS) distilled from the firing semantics of an OCPR net.

Definition 4.16 (labelled transition system). A labelled transition system (simply, LTS) is a tuple $\langle S, \Lambda, R, s_0 \rangle$, where $S$ is a set of states, $\Lambda$ is a set of labels, $R \subseteq S \times \Lambda \times S$ is the transition relation, and $s_0 \in S$ is the initial state. If $p, q \in S$ and $\alpha \in \Lambda$, then $\langle p, \alpha, q \rangle \in R$ is written as $p \xrightarrow{\alpha} q$.

The introduction of a LTS is inspired to the theory of reactive systems [43]. This meta-theory suggests guidelines for deriving a LTS from an unlabelled one, choosing a set of labels with suitable requirements of minimality. In the setting of OCPR nets, the reduction relation is given by $\langle \rangle$, and a firing is allowed if all the preconditions of a transition are satisfied. Thus, intuitively, the minimal context that allows a firing just adds the tokens needed for that firing.

Definition 4.17 (transition relation). Let $\mathcal{N}$ be an OCPR net, and let $\Lambda = \{\tau\} \cup (\{+\} \times P_N) \cup \{-\} \times C_N$ be a set of labels, ranged over by $l$. The transition relation for $\mathcal{N}$ is the relation $R_\mathcal{N}$ inductively generated by the set of inference rules below

\[
\begin{align*}
\frac{o \in \text{Op}(\mathcal{N}) \setminus (M \cup \{f\})}{M \xrightarrow{o}_\mathcal{N} M \cup \{o\}} & \quad \quad \frac{f \in M}{M \xrightarrow{f}_\mathcal{N} M \setminus \{f\}} & \quad \quad \frac{M \upharpoonright M'}{M \xrightarrow{f}_\mathcal{N} M'}
\end{align*}
\]

where $M \xrightarrow{l}_\mathcal{N} M'$ means that $\langle M, l, M' \rangle \in R_\mathcal{N}$, and $i, f$ denote the initial and the final place of $\mathcal{N}$, respectively.
Thus, a context may add tokens in open places (if not already inhabited and
with the exception of the final place $f$, as stated by $Op(N) \setminus (M \cup \{f\})$) in order
to perform a firing, as represented by the transition $\overset{+o}{\rightarrow}_N$. Similarly, a context may
consume tokens from the final place $f$. A context cannot interact with the net in
other ways, as the initial place $i$ can be used by the context only as a post condition,
as well as all the other open places are data places whose tokens can be read but
not consumed. Instead, $\tau$ transitions represent internal firing steps, i.e., steps that
do not need any additional token from the environment.

The theory of reactive systems ensures that, for a suitable choic e of labels, the
(strong) bisimilarity on the derived LTS is a congruence [43]. However, often that
bisimilarity does not coincide with the saturated one. In the case at hand, we
introduce a notion of weak bisimilarity, abstracting away from the number of steps
performed by nets, that indeed coincides with the saturated one.

**Definition 4.18 (weak bisimulation).** A symmetric relation $R \subseteq MN \times MN$
is a weak bisimulation if whenever $(N, M) R (N', M')$ then

- $O_N = O_{N'}$ and $\text{Obs}(N, M) = \text{Obs}(N', M')$,
- $M \overset{+o}{\rightarrow}_N M_1$ implies $M' \overset{+o}{\rightarrow}_{N'} M'_1 \in (N, M) R (N', M')$,
- $M \overset{-f}{\rightarrow}_N M_1$ implies $M' \overset{-f}{\rightarrow}_{N'} M'_1 \in (N, M) R (N', M')$, and
- $M \overset{\tau}{\rightarrow}_N M_1$ implies $M' \overset{\tau}{\rightarrow}_{N'} M'_1 \in (N, M) R (N', M')$.

The union of all weak bisimulations is called weak bisimilarity ($\approx_W$).

The key theorem of this Subsection is stated below.

**Theorem 4.1** $\approx_S = \approx_W$.

Thus, in order to prove that two OCPR nets are bisimilar, it suffices to exhibit
a weak bisimulation between the states of the two nets that includes the pair of empty markings. Most importantly, though, this verification can be automatically performed, since the set of possible states of an OCPR net are finite. Hence, the result below immediately follows.

**Corollary 4.1** $\approx_S$ is decidable.
4.3. REASONING ON SERVICE BEHAVIOUR

4.3.2 Checking behavioural properties of Web services

The behavioural analysis provides two different analyses of service behaviour, namely, it executes a lock analysis when a functional query is received, while it executes a bisimilarity analysis when a behavioural query is received. The lock analysis checks whether a given service generates the requested outputs without locking, while the bisimilarity analysis verifies the behavioural equivalence between such a service and the (behavioural) client request. In other words, the lock analysis animates the OCPR net modelling the given service to check whether it reaches the final state in which the final control place of the net as well as the data places corresponding to the query outputs are inhabited by a token. The bisimilarity analysis verifies whether the OCPR net modelling a given service behaves as the OCPR modelling the client request (e.g., whether the nets evolve in the same (observable) states), if animated by inhabiting the same (open) places.

In this Subsection we introduce a (trivial) algorithm performing the lock analysis, while we employ an existing algorithm to perform the bisimilarity analysis. Both algorithms execute on the labelled transition systems derived from OCPR nets and characterised by Definition 4.17.

The rest of this Subsection is hence organised as follows. We first present the OCPR2LTS function, which constructs the LTS of a given OCPR net. Next, we present the functions CheckMayTermination and CheckMustTermination, which perform the lock analysis with different levels of quality. Finally, we introduce an existing algorithm which checks whether it is possible to define a bisimilarity relation between two given labelled transition systems.

Constructing labelled transition systems from OCPR nets

Definition 4.17 provides three rules for deriving a labelled transition system from the firing semantics of a OCPR net. A state of the labelled transition system is a marking (viz., a set of places) of the OCPR net. It is possible to move from a state to another one by inhabiting an open place of the net, by removing a token from the final control place \( f \) of the net, and by executing an internal transition (viz., not requiring additional external input) of the net, namely, exactly as per the three rules of Definition 4.17. In order to provide a general theory, we defined such rules for arbitrary (OCPR) nets. Yet, in this Subsection we introduce and discuss algorithms executing on labelled transition systems derived from OCPR nets only constructed according to the encoding (from OWL-S descriptions to OCPR nets) presented in Subsection 4.1.4. We note, then, that the second rule of Definition 4.17 is not strictly necessary for these kind of nets. Indeed, as one can observe for instance in Figure 4.5 of Subsection 4.1.4, after removing the token from the final control place \( f \) of a net, no transition can fire any more. It is also worth noting that we do not consider interleaved multi-instances of the same service profile, thus checking for the bisimilarity (and termination) of single instances of service profiles.
Hence, the first rule of Definition 4.17 can be simplified as well, since once the initial control place $i$ of a net has been emptied, it can no longer be inhabited any more. In such a case, two nets $\mathcal{N}, \mathcal{N}'$ are bisimilar if $(\mathcal{N}, \{i\}) \approx_{\mathcal{S}} (\mathcal{N}', \{i\})$ (while in the general theory $\mathcal{N}, \mathcal{N}'$ are bisimilar if $(\mathcal{N}, \emptyset) \approx_{\mathcal{S}} (\mathcal{N}', \emptyset)$, as stated in Definition 4.15).

The rules of Definition 4.17 can be hence simplified as follows:

$$\frac{o \in Op(\mathcal{N}) \setminus (M \cup \{i, f\})}{M \xrightarrow{t, o_{\mathcal{N}}} M \cup \{o\}} \quad \frac{M \xrightarrow{\tau_{\mathcal{N}}} M'}{M \xrightarrow{\tau} M'}$$

The pseudo-code for deriving a labelled transition system from the firing semantics of a given OCPR net is listed in Figure 4.19. Yet, before presenting the pseudo-code, we roughly sketch – through a toy example – how to construct a labelled transition system from a given OCPR net.

Let us consider the OCPR net $\mathcal{N}_1$ illustrated in the left part of Figure 4.17. We have to construct the labelled transition system which represents all the possible ways the environment can interact with the net. A state of the labelled transition system denotes a marking (viz., a set of places) of the net. Hence, according to the simplified version of the rules of Definition 4.17, for each state of LTS:

1. we can insert a token in an open data place $o$ of the net, if $o$ is not inhabited,
2. we can fire one of the enabled transitions of the net (viz., the $\tau$ transitions), if any.

![Figure 4.17: Two sample OCPR nets.](image)

The initial state of the LTS is the set $\{i\}$ (viz., the marking of the net $\mathcal{N}_1$ in which only the initial control place $i$ is inhabited). First, we note that, initially, no

---

4As a consequence, the cases $M \xrightarrow{t} M_1$ and $M \xrightarrow{t, o_{\mathcal{N}}} M_1$, for $o = i$, do not occur in Definition 4.18, thus originating a notion of bisimulation that, in general, is weaker than the one in Definition 4.18.
4.3. REASONING ON SERVICE BEHAVIOUR

A transition is enabled in \( \mathcal{N}_1 \), hence no \( \tau \) transition can fire in the initial state \( \{i\} \). Yet, we can insert a token either in the data place \( A \) or in the data place \( B \) of \( \mathcal{N}_1 \). Thus, we add to the LTS the new two states \( \{i, A\} \) and \( \{i, B\} \), which can be reached from the initial state \( \{i\} \) by inhabiting the places \( A \) and \( B \), respectively. As one can observe in Figure 4.18, this is graphically represented by two arcs, the first labelled with \( A \) and connecting \( \{i\} \) to \( \{i, A\} \) and the second labelled by \( B \) and connecting \( \{i\} \) to \( \{i, B\} \). Consider now the state \( \{i, A\} \). We note that no internal transition is enabled in \( \mathcal{N}_1 \) in such a state, while we can only insert a token in the data place \( B \). Hence, we add the new state \( \{i, A, B\} \) to the LTS, and we connect the state \( \{i, A\} \) to \( \{i, A, B\} \) with an arc labelled by \( B \). Let us consider the state \( \{i, A, B\} \). We can not add tokens in the open places of the net, since \( A, B \) have been already inhabited, yet, internal transitions \( t_1 \) and \( t_2 \) of \( \mathcal{N}_1 \) are both enabled, since their input data (viz., \( A, B \) and control places (viz., \( i \)) are included in the current marking \( \{i, A, B\} \) of the net. We add the states \( \{A, B, c_1\} \) and \( \{A, B, c_2\} \) to the LTS (note that \( c_1 \) and \( c_2 \) are the internal control places of \( \mathcal{N}_1 \)), which can be reached from the state \( \{i, A, B\} \) by firing the \( \tau \) transitions \( t_1 \) and \( t_2 \), respectively. Note, indeed, the arc labelled by \( \tau \) and connecting \( \{i, A, B\} \) to \( \{A, B, c_1\} \) as well as the arc labelled by \( \tau \) and connecting \( \{i, A, B\} \) to \( \{A, B, c_2\} \) in Figure 4.18. Next, let us consider the state \( \{A, B, c_1\} \). Again, we can not add tokens in the places \( A, B \), yet, the internal transition \( t_3 \) can fire. The new state \( \{A, B, f\} \) is inserted to the LTS together with the arc labelled by \( \tau \) and connecting \( \{A, B, c_1\} \) to \( \{A, B, f\} \). \( \{A, B, f\} \) is a final state, since no internal transition can fire in the marking \( \{A, B, f\} \) as well as no open place can be inhabited further. Finally, consider, for example, the state \( \{i, B\} \). In such a marking, we can either add a token in the open place \( A \) and reach the already analysed state \( \{i, A, B\} \), or fire the internal transition \( t_2 \) and reach the new state \( \{B, c_2\} \). Hence, we add the state \( \{B, c_2\} \) to the LTS together with two arcs, the first connecting \( \{i, B\} \) to \( \{i, A, B\} \) and labelled by \( A \), and the second connecting \( \{i, B\} \) to \( \{B, c_2\} \) and labelled by \( \tau \). We continue by generating new LTS states until all the reached states are final states (or no new state can be reached). The complete LTS derived from the sample net \( \mathcal{N}_1 \) is illustrated in Figure 4.18.

![Figure 4.18: The LTS derived from the firing semantics of the net \( \mathcal{N}_1 \).](image)

The OCPR2LTS function described in Figure 4.19 derives a labelled transition system from a given OCPR net by implementing the behaviour roughly discussed
in the example above.

In the pseudo-code of Figure 4.19 we denote a labelled transition system as a pair \(LT S = (S, R)\), where \(S\) and \(R\) are the set of the states and the set of the rules of \(LT S\), respectively. A state \(s \in S\) is a set of places, while a rule \(r \in R\) is a triple \((s, l, t)\), where \(s \in S\) denotes the source state, \(l\) the label and \(t \in S\) the target state of the rule, respectively. A label \(l\) is a pair \((a, Q)\), where \(a\) can be \(\tau\) or an open data place of the net \(N\) (viz., \(a = \{\tau\} \cup Op(N) \setminus \{i, f\}\)), and \(Q\) denotes the open places of the target state \(t\) (viz., \(Q = t \cap Op(N)\)). It is important to note that such a definition of labels is needed in order to use the bisimulation algorithm in [31, 32] for checking the weak bisimulation on \(LT S\)s accordingly to Definition 4.18. We will better clarify this point at the end of this Subsection, after briefly introducing the algorithm in [31, 32]. For instance, Figure 4.21 illustrates the same labelled transition system of Figure 4.18 (derived from the sample net \(N_1\) in Figure 4.17), but for labels defined as pairs. Note for example the arc labelled by \(A\) and connecting the state \(\{B, c_2\}\) to the state \(\{A, B, c_2\}\) in the \(LT S\) of Figure 4.18. It corresponds to the arc labelled by \((A, \{A, B\})\) which connects \(\{B, c_2\}\) to \(\{A, B, c_2\}\) in the \(LT S\) of Figure 4.21. In particular, such an arc can be expressed by the rule \(\langle\{B, c_2\}, (A, \{A, B\}), \{A, B, c_2\}\rangle\).

---

1. \(OCPR2LTS(OCPR N = \langle N, O \rangle)\)
2. \(S = \emptyset; R = \emptyset; \LT S = (S, R);\)
3. Add the initial state \(\{i\}\) to \(S\);
4. \(statesToBeExplored = \{\{i\}\};\)
5. \(\textbf{while} (statesToBeExplored \neq \emptyset) \textbf{do}\)
6. \(sourceState = \text{extract}(statesToBeExplored);\)
7. \(\textbf{forall} \text{ open data places } o \text{ in } (Op(N) \setminus (sourceState \cup \{i, f\})) \textbf{do}\)
8. \(targetState = sourceState \cup \{o\};\)
9. \(\textbf{if} (targetState \notin S) \textbf{then}\)
10. \(\text{Add } targetState \text{ to } S;\)
11. \(statesToBeExplored = statesToBeExplored \cup \{targetState\};\)
12. \(\text{Add } \langle sourceState, (o, targetState \cap Op(N)), targetState \rangle \text{ to } R;\)
13. \(\textbf{forall} \text{ transition } t \text{ in } T_N \textbf{ do}\)
14. \(\textbf{if } ((^*t \cup t^\circ) \subseteq sourceState) \land ((sourceState \cap t^\circ) \subseteq t^\circ) \textbf{ then}\)
15. \(targetState = (sourceState \setminus t^\circ) \cup t^\circ \cup t^*;\)
16. \(\textbf{if } (targetState \notin S) \textbf{ then}\)
17. \(\text{Add } targetState \text{ to } S;\)
18. \(statesToBeExplored = statesToBeExplored \cup \{targetState\};\)
19. \(\text{Add } \langle sourceState, (\tau, targetState \cap Op(N)), targetState \rangle \text{ to } R;\)
20. \(\textbf{return } \LT S;\)

---

Figure 4.19: Pseudo-code for deriving a \(LT S\) from an OCPR net.
4.3. REASONING ON SERVICE BEHAVIOUR

The \texttt{OCPR2LTS} function takes as input an OCPR net $N = (N, O)$ (line 1), where $N = (C_N, D_N, T_N, F_N, G_N)$ is the underlying CPR net, and $O = (i, f, OD)$ in the outer interface (viz., the initial control place, the final control place and the open data places) of $N$ (Definition 4.5). Firstly, \texttt{OCPR2LTS} initialises the labelled transition system $LTS = (S, R)$ to be generated (line 2), and adds the initial state \{i\} to it (line 3). \texttt{OCPR2LTS} stores the $LTS$ states still to be analysed in the statesToBeExplored set. Initially, the state \{i\} is hence inserted to statesToBeExplored (line 4). Then, while there exists a state to be analysed (viz., statesToBeExplored $\neq \emptyset$) (line 5), \texttt{OCPR2LTS} extracts a state sourceState from statesToBeExplored (line 6), and determines all the states directly reachable (viz., by means of a single rule) from sourceState, thus generating the necessary rules (lines 7-19). In particular, \texttt{OCPR2LTS} first computes the open data places of the net where it is still possible to add a token (line 7). Next, for each selected open data place $o$, \texttt{OCPR2LTS} determines the targetState reached from the sourceState by inhabiting $o$ (line 8). If targetState does not already belong to the states $S$ of LTS (line 9), it adds targetState to $S$ (line 10) and to the statesToBeExplored set (line 11). Then, also the rule $\langle$sourceState, $(o, targetState \cap Op(N))$, targetState$\rangle$ is added to LTS (line 12). Next, for each transition $t$ of $N$ (line 13), \texttt{OCPR2LTS} checks whether $t$ can fire (according to Definition 4.4) in the current state sourceState (line 14). If so, it computes the targetState reached by firing $t$ (line 15), and, if targetState does not belong to $S$ (line 16), adds targetState to LTS (line 17) and to the statesToBeExplored set (line 18). \texttt{OCPR2LTS} also adds the rule $\langle$sourceState, $(\tau, targetState \cap Op(N))$, targetState$\rangle$ to LTS (line 19). Finally, the resulting labelled transition system is returned (line 20).

It is worth noting that the \texttt{OCPR2LTS} function always terminates, since the number of the states (viz., markings) that an OCPR net can reach is finite. When there is no new state to be analysed (line 5), \texttt{OCPR2LTS} terminates (line 20).

Lock analysis

We distinguish two cases of successful termination of a service. Namely a service:

- \textit{may-terminate} – if there exists (at least) a terminating behaviour of the service which generates the requested outputs without locking,

- \textit{must-terminate} – if all the possible behaviour of the service terminate and generate the requested outputs without locking.

As a consequence, we provide two different lock analyses, the first checking the \textit{may}-termination of a service, and the second checking the \textit{must}-termination of a service. In Figure 4.20, we present the pseudo-code of the (trivial) functions CheckMayTermination and CheckMustTermination, which respectively check the may-termination and the must-termination of a service. Such functions perform the lock analyses by analysing the labelled transition system derived from an OCPR net,
which, in turn, models the full behaviour of a service. In particular, \texttt{CheckMayTermination} and \texttt{CheckMustTermination} are designed to cope with LTSs derived from OCPR nets that have been generated from OWL-S process models according to the encoding presented in Subsection 4.1.4. Intuitively, each trace of a labelled transition system corresponds to a possible behaviour of a service. Hence, verifying whether a behaviour of a service \textit{may}-terminate by generating the requested outputs corresponds to check whether there exists (at least) a trace terminating in a final state which contains the data place modelling the requested outputs and the final control place of the underlying net. Similarly, verifying the \textit{must}-termination of a service consists of checking whether \textit{each} trace terminates in a final state which contains the data place modelling the requested outputs and the final control place of the underlying net.

In the pseudo-code of Figure 4.20, $L = \langle S, R \rangle$ denotes a labelled transition system with a set of states $S$ and a set of rules $R$, as described previously. Moreover $\text{Out}(Q)$ denotes the outputs of a client query $Q$, while $i$ and $f$ respectively denote the initial control place and the final control place of the OCPR net underlying $L$.

1. \texttt{CheckMayTermination}(LTS $L = \langle S, R \rangle$, Query $Q$, Place $f$)
2. \hspace{1em} $M = \{ m : (m \in S) \land (\exists l, r : \langle m, l, r \rangle \in R) \}$;
3. \hspace{1em} \texttt{forall final states $m$ in $M$ do}
4. \hspace{2em} \texttt{if ($\text{Out}(Q) \cup \{ f \} \subseteq m$) then return true;}
5. \hspace{1em} \texttt{return false;}

6. \texttt{CheckMustTermination}(LTS $L = \langle S, R \rangle$, Query $Q$, Place, $i$, Place $f$)
7. \hspace{1em} $M = \{ m : (m \in S) \land (\exists l, r : \langle m, l, r \rangle \in R) \}$;
8. \hspace{1em} \texttt{forall final states $m$ in $M$ do}
9. \hspace{2em} \texttt{if ($\text{Out}(Q) \cup \{ f \} \not\subseteq m$) then return false;}
10. \hspace{1em} \texttt{return $\neg$(FindLoops($L, \{ i \}, \emptyset$));}

11. \texttt{boolean FindLoops}(LTS $L = \langle S, R \rangle$, State $s$, set $\text{visitedStates}$)
12. \hspace{1em} \texttt{if ($s \in \text{visitedStates}$) then return true}
13. \hspace{1em} \texttt{else}
14. \hspace{2em} $T = \{ t : \exists l : \langle s, l, t \rangle \in R \}$;
15. \hspace{2em} \texttt{if ($T \neq \emptyset$) then}
16. \hspace{3em} \texttt{forall state $t$ in $T$ do}
17. \hspace{4em} \texttt{if (FindLoops($L, t, \text{visitedStates} \cup \{ s \}$) then return true;}
18. \hspace{1em} \texttt{return false;}

---

Figure 4.20: Pseudo-code for the lock analysis.

The \texttt{CheckMayTermination} function takes as input a labelled transition system $L = \langle S, R \rangle$, a \textit{functional} client request $Q$, and the final control place $f$ of the OCPR
net from which \( L \) has been derived (line 1). First, it determines the set \( M \) of the final states (viz., the states with no outgoing arcs) of \( L \) (line 2). Then, for each final state \( m \in M \) (line 3), it checks whether \( m \) contains the requested outputs and the final control place \( f \) of the underlying net (line 4). If so, \texttt{CheckMayTermination} exits, since the \textit{may}-termination check succeeds. Otherwise if none of the final states of \( L \) contain both the requested outputs and the final control place, the \textit{may}-termination check fails (line 5).

The \texttt{CheckMustTermination} function takes as input a labelled transition system \( L = \langle S, R \rangle \), a \textit{functional} client request \( Q \), the initial control place \( i \), and the final control place \( f \) of the OCPR net underlying \( L \) (line 6). \texttt{CheckMustTermination} computes the set \( M \) of the final states of \( L \) (line 7), and for each state \( m \in M \) (line 8) it checks whether \( m \) contains the requested outputs and the final control place \( f \) of the underlying net (line 9). If this is not the case, \texttt{CheckMustTermination} exits (line 9), since the \textit{must}-termination check fails. Otherwise (line 10), \texttt{CheckMustTermination} checks the existence of cycles in \( L \) (viz., livelocks) by means of the supporting function \texttt{FindLoops} (presented in Figure 4.20). If \( L \) contains cycles (viz., \texttt{FindLoops} returns true), \texttt{CheckMustTermination} returns false, otherwise (viz., \texttt{FindLoops} returns false), \texttt{CheckMustTermination} succeeds.

The \texttt{FindLoops} function takes as input a labelled transition system \( L = \langle S, R \rangle \), a state \( s \) (initially the initial state \( \{i\} \) of \( L \)), and a set \texttt{visitedStates} of the visited states of \( L \) (initially empty). \texttt{FindLoops} recursively visits (in a depth-first order) all the possible evolutions (viz., traces) of \( L \). \texttt{FindLoops} immediately exits (line 12) by returning true, if a trace reaches an already visited state (viz., \( L \) contains a cycle), otherwise, after visiting \( L \), it terminates by returning false.

For example, consider the (choice) service which inputs \( A \) and \( B \) and does something, whose behaviour is modelled by the OCPR net \( N_1 \) depicted in the left part of Figure 4.17. To check the \textit{must}-termination of such a service, we first generates the labelled transition system derived from the OCPR net \( N_1 \) by employing, for instance, the OCPR2LTS function described in Figure 4.19. The resulting labelled transition system is illustrated in Figure 4.18. Then, we check whether all the final states of such a LTS contain the final control place \( f \) of the underlying OCPR net. As one may observe in Figure 4.18, the final state \( \{A, B, f\} \) of the LTS contains \( f \), and moreover, the LTS does not contain cycles. Hence, the choice service \textit{must}-terminate (and obviously \textit{may}-terminate, as well).

\textbf{Bisimilarity analysis}

The bisimilarity analysis employs the notion of bisimilarity introduced in Subsection 4.3.1 to check whether two services feature the same (externally observable) behaviour. Important issues, such as, for example, to verify the equivalence of two different versions of a service, to verify whether a service can safely replace another service, to check whether a service implements a given specification, and to
check whether a (composite) service matches a given (behavioural) client request, can be suitably addressed by employing such a notion of service equivalence. In particular, the behavioural analysis implements the bisimilarity analysis to verify whether a composition of services (e.g., selected by the functional analysis) matches the behaviour that a client specifies in a behavioural request.

Our relying on the concept of bisimilarity allows us to benefit from the wealth of tools and algorithms developed so far. Indeed, as discussed in Subsection 4.3.1 (see Definitions 4.17 and 4.18), we can check bisimilarity by constructing a finite labelled transition system and then verifying (weak) bisimilarity there. This is the case, e.g., of [31, 32], where an algorithm for verifying the bisimilarity of two labelled transition systems was introduced.

The key ingredient of the algorithm in [31, 32] is the notion of synchronous product $L_1 \times L_2$ between two labelled transition systems $L_1$ and $L_2$. Then, the existence of a bisimilarity\(^5\) relation $R$ between $L_1$ and $L_2$ depends on a simple criterion which must hold on the execution sequences of $L_1 \times L_2$. Intuitively, the product $L_1 \times L_2$ can be defined as follows:

- a state $(s_1, s_2)$ of $L_1 \times L_2$ can perform a transition labelled by an action $a$ if and only if the state $s_1$ in $L_1$ and the state $s_2$ in $L_2$ can perform a transition labelled by $a$, otherwise

- if only one of the two states $s_1, s_2$ can perform a transition labelled by $a$, then the product $L_1 \times L_2$ has a transition from $(s_1, s_2)$ to the sink state $\text{fail}$.

We include hereafter the formal definition of synchronous product of two LTSs, as introduced in [31, 32].

**Definition 4.19 (synchronous product of LTSs).** Given two LTSs $L_1 = \langle S_1, \Lambda_1, R_1, s_{01} \rangle$ and $L_2 = \langle S_2, \Lambda_2, R_2, s_{02} \rangle$, the synchronous products $L_1 \times L_2$ is the LTS $L = \langle S, \Lambda, R, (s_{01}, s_{02}) \rangle$, where $S \subseteq (S_1 \times S_2) \cup \{\text{fail}\}$, $\Lambda = (\Lambda_1 \cap \Lambda_2) \cup \{\phi\}$, and $R \subseteq S \times \Lambda \times S$, with $\phi \notin (\Lambda_1 \cup \Lambda_2)$ and $\text{fail} \notin (S_1 \cap S_2)$.

$R$ and $S$ are defined as the smallest sets obtained by the applications of the following rules.

\[
\begin{align*}
(s_{01}, s_{02}) & \in S \\
(s_1, s_2) & \in S, \quad \text{Act}_\Pi(s_1) = \text{Act}_\Pi(s_2), \quad s_1 \xrightarrow{\lambda_{R_1}} s'_1, s_2 \xrightarrow{\lambda_{R_2}} s'_2 \\
\{ (s'_1, s'_2) \} & \in S, \{ (s_1, s_2) \xrightarrow{\lambda_R} (s'_1, s'_2) \} \in R \\
(s_1, s_2) & \in S, s_1 \xrightarrow{\lambda_{R_1}} s'_1, R_2[s_2] = \emptyset \\
\{ \text{fail} \} & \in S, \{ (s_1, s_2) \xrightarrow{\phi_R} \text{fail} \} \in R
\end{align*}
\]

\(^{5}\)The algorithm in [31, 32] can be employed to check the existence of a similarity relation between two LTSs, as well. Yet, we focus here on its capability of determining bisimilarity relations.
where \( R^\lambda[s] = \{ s' \in S \mid s \xrightarrow{\tau} R s' \} \), and \( \text{Act}_\Pi(s) = \{ \lambda \in \Pi \mid \exists s' : s \xrightarrow{\lambda} s' \} \), with \( \Pi \) a family of disjoint languages on \( \Lambda \).

It is worth noting that the strong bisimulation is defined by \( \Pi = \{ \{ a \} \mid a \in \Lambda \} \), while weak bisimulation — the one which we consider in this thesis — is defined as \( \Pi = \{ \tau^* a \mid a \in \Lambda \land a \neq \tau \} \), where \( \tau^* a \) denotes a (possibly empty) sequence of internal action \( \tau \) followed by an action \( a \in \Lambda \). Roughly, in order to check the non existence of a bisimulation relation \( \mathfrak{R} \) between two LTSs \( L_1 \) and \( L_2 \), it is sufficient to check whether there exists an execution sequence \( \sigma \) of the product \( L = L_1 \times L_2 \) which contains the state \( \text{fail} \). If such a sequence \( \sigma \) exists, then \( L_1 \) is not bisimilar to \( L_2 \), otherwise \( L_1 \) is bisimilar to \( L_2 \).

The algorithm in \([31, 32]\) verifies the existence of \( \sigma \) by performing depth-first searches on the product \( L = L_1 \times L_2 \) (which is constructed “on the fly”). Basically, the algorithm associates a bit-array \( M \) of size \( |T[s_1]| + |T[s_2]| \) to each state \( (s_1, s_2) \in L \), where \( T[s_i] = \{ s'_i \in S_i \mid (s_i, a, s'_i) \in T_i \} \) (viz., \( T[s_i] \) is the direct successors of \( s_i \) in the LTS \( L_i = (S_i, A_i, R_i, s_{i0}) \)). During the analysis of each successor \( (s'_1, s'_2) \) of \( (s_1, s_2) \), computed by employing the rules of Definition 4.19, when it happens that \( (s'_1, s'_2) \in \mathfrak{R} \) then \( M[s'_1] \) and \( M[s'_2] \) are set to 1. Consequently, when all the successors of \( (s_1, s_2) \) have been analysed, \( (s_1, s_2) \in \mathfrak{R} \) if and only if all the elements of \( M \) have been set to 1.

In other words, the algorithm in \([31, 32]\) visits the states of the product LTS \( L_1 \times L_2 \) in a depth-first order, thus starting to check the bisimilarity from the leaf states (viz., the states with no outgoing arcs) and then continuing backwards, until reaching the initial state. Hence, intuitively, when the depth-first visit of the product LTS terminates, the bit-array \( M \) associated to the initial state of the product LTS is checked. If all the element of \( M \) have been set to 1, the LTSs \( L_1 \) and \( L_2 \) are bisimilar.

The algorithm in \([31, 32]\) is proved to be correct and always terminating. Furthermore, its time complexity is \( O(n^2) \), where \( n \leq n_1 \times n_2 \) is the number of the states of the product LTS \( L_1 \times L_2 \).

It is worth observing that the rules of Definition 4.19 does not check whether two states \( s_1, s_2 \) contain the same observable places, as instead required by our notion of weak bisimilarity (Definition 4.18). For example, consider a LTS \( L_3 \) with two states \( \{ A \} \) and \( \{ A, B \} \), where \( \{ A \} \) can reach \( \{ A, B \} \) by performing a \( \tau \) transition. Then, consider a LTS \( L_4 \) with an initial state \( \{ A \} \) that reaches a state \( \{ A, C \} \) through a \( \tau \) transition. According to Definition 4.19, in the product LTS \( L_3 \times L_4 \) the state \( \{ A, B \}, \{ A, C \} \) is a successor of the initial state \( \{ \{ A \}, \{ A \} \} \). Moreover, the single execution sequence of the product LTS does not contain the state \( \text{fail} \), and \( L_3 \) and \( L_4 \) are hence bisimilar. Yet, according to Definition 4.18, \( L_3 \) and \( L_4 \)
are not behaviourally equivalent, since the state \{A, B\} in \(L_3\) does not contain the same observable places of the state \{A, C\} in \(L_4\).

In order to use the algorithm in [31, 32] to verify the weak bisimulation on LTSs accordingly to Definition 4.18, it is necessary to employ labels defined as pairs \((a, Q)\), where \(a \in \Lambda\) and \(Q\) denotes a set of observable places. More precisely, if a state \(s\) can reach a state \(t\) with a transition labelled by \((a, Q)\), \(Q\) denotes the observable places of \(t\). For example, consider again the LTSs \(L_3\) and \(L_4\) of the example above. In \(L_3\), the state \(\{A\}\) can reach the state \(\{A, B\}\) with a transition labelled by \((\tau, \{A, B\})\), while in \(L_4\) the state \(\{A\}\) can reach the state \(\{A, C\}\) with a transition labelled by \((\tau, \{A, C\})\). In such a case, the successor of the initial state \((\{A\}, \{A\})\) of the product LTS \(L_3 \times L_4\) is the state \textit{fail}. Correctly, \(L_3\) and \(L_4\) does not result bisimilar.

We do not include here the full pseudo-code of the algorithm, which can be found in [31, 32], yet, we illustrate an example. In particular, we want to check whether the OCPR nets \(N_1\) and \(N_2\) depicted in Figure 4.17 are behaviourally equivalent. The labelled transition systems \(L_1\) and \(L_3\) derived from the firing semantics of \(N_1\) and \(N_2\) are presented in Figures 4.21 and 4.22, respectively.

![Figure 4.21: The LTS derived from the firing semantics of the net \(N_1\).](image1)

![Figure 4.22: The LTS derived from the firing semantics of the net \(N_2\).](image2)

The product LTS of \(L_1\) (Figure 4.21) and \(L_2\) (Figure 4.22) is depicted in Figure 4.23. Obviously, the initial state of the product LTS is the state \((\{i\}, \{i\})\). Then, we observe that both the initial states \(\{i\}\) of \(L_1\) and \(\{i\}\) of \(L_2\) can perform two transitions labelled by \((A, \{i, A\})\) and \((B, \{i, B\})\) and reach the states \(\{i, A\}\), \(\{i, B\}\) of \(L_1\) and \(\{i, A\}\), \(\{i, B\}\) of \(L_2\), respectively. Hence, the successors of \((\{i\}, \{i\})\) are the two states \((\{i, A\}, \{i, A\})\) and \((\{i, B\}, \{i, B\})\), which the state \((\{i\}, \{i\})\) can reach.
4.3. REASONING ON SERVICE BEHAVIOUR

through two transitions labelled by \( (A, \{i, A\}) \) and \( (B, \{i, B\}) \), respectively. Let us consider the state \( (\{i, A\}, \{i, A\}) \) of the product LTS. We note that both the states \( \{i, A\} \) of \( L_1 \) and \( \{i, A\} \) of \( L_2 \) can perform a transition labelled by \( (B, \{i, A, B\}) \) and reach the states \( \{i, A, B\} \) of \( L_1 \) and \( \{i, A, B\} \) of \( L_2 \), respectively. The state \( (\{i, A, B\}, \{i, A, B\}) \) is hence the successor of \( (\{i, A\}, \{i, A\}) \), which can perform a transition labelled by \( (B, \{i, A, B\}) \) in the product LTS. The state \( \{i, A, B\} \) of \( L_2 \) (Figure 4.22) can perform two \( \tau \) transitions and reach the state \( \{A, B, f\} \), while the state \( \{i, A, B\} \) of \( L_1 \) (Figure 4.18) can reach the state \( \{A, B, f\} \) by means of two distinct \( \tau \) transitions. Then, the state \( (\{i, A, B\}, \{i, A, B\}) \) of the product LTS has one successor, viz., \( (\{A, B, f\}, \{A, B, f\}) \), which can be reached through two distinct transitions labelled by \( (\tau, \{A, B, f\}) \). The full product LTS is illustrated in Figure 4.23. As one may note, there is no execution sequence of the product LTS which contains the state \textit{fail}, hence the LTS \( L_1 \) and \( L_2 \) are bisimilar. Namely, The OCPR nets \( N_1 \) and \( N_2 \) in Figure 4.17 are behaviourally equivalent.

![Figure 4.23: The product of the LTSs derived from nets \( N_1 \) and \( N_2 \).](image)

Finally, as a corollary of the soundness of the algorithm in [31, 32] (used by the bisimilarity analysis to check services’ equivalence), we can state the following proposition.

**Proposition 4.2** Let \( S \) be a (composite) service returned by the bisimilarity analysis for a given behavioural query \( Q \). Then, \( S \) features the same (externally observable) behaviour of \( Q \).

4.3.3 An example

We now illustrate a sample scenario concerning banking systems that allows us to discuss the issues of \textit{behavioural matching} as well as of \textit{service replaceability}. Let us consider again the second example previously introduced in Subsection 3.2.4. The \textit{CreditPortal} (Figure 2.3) provider (i.e., the bank) wants externalise the \textit{CreditPortal} section which computes customer rating. We depicted in Figure 3.15 the \textit{behavioural} query (i.e., the OWL-S process model of the desired service) that the provider submitted to our discovery framework. As detailed in Subsection 3.2.4, the functional
analysis returned four successful profiles – viz., \( \{RO_1\} \), \( \{RT_1\} \), \( \{RT_2\} \), \( \{RT_3\} \), corresponding to different behaviour of the RatingOne and RatingTwo services – each of them satisfying the functional requirements of the client query. Hence, the behavioural analysis has now to check, for each service, whether it behaves as the behavioural query expects. In other words, the behavioural analysis is asked to verify whether the sub-service of CreditPortal which evaluates the customer rating can be replaced by RatingOne or RatingTwo, not altering the whole CreditPortal application.

First, the behavioural analysis translates the client query (viz., the OWL-S process model of Figure 3.15) into an OCPR net, as depicted in Figure 4.24. Let \( N_Q \) denotes such a net.

![Figure 4.24: The OCPR net representation of the client query.](image)

Next, it considers the OCPR net \( N_{RO} \) modelling the behaviour of the RatingOne service (Figure 4.12). Yet, before equating the two nets, the behavioural analysis applies the hiding context \( \nu_{A, O_{RO}} \) to \( N_{RO} \), where \( A = \{firstRating, secondRating, thirdRating\} \), and \( O_{RO} \) denotes the interface of \( N_{RO} \). Indeed, data places firstRating, secondRating and thirdRating do not belong to the interface of \( N_Q \) (i.e., the provider is just interested in observing the final rating, not the intermediate ones), hence the behavioural analysis does not observe them. Now, the behavioural analysis equates \( N_Q \) and \( \nu_{A, O_{RO}}[N_{RO}] \), and the two nets result equivalent, in particular, they are structurally different, but externally indistinguishable. Hence, RatingOne can be employed to properly replace the sub-service of CreditPortal evaluating the customer rating.

Let us now consider the OCPR net \( N_{RT} \) modelling the behaviour of the RatingTwo service (Figure 4.15). Again, the behavioural analysis applies the hiding context \( \nu_{A, O_{RT}} \) to \( N_{RT} \), where \( O_{RT} \) denotes the interface of \( N_{RT} \) (i.e., firstRating, secondRating, thirdRating become close places). Then, it equates \( N_Q \) and \( N_{RT} \), which result equivalent.

As we described in Subsection 2.1.2, RatingOne firstly computes three separate evaluations of the customer and then it returns an average rating, RatingTwo computes the customer rating and only if necessary it performs a second and possibly a third evaluation of the customer. RatingTwo may be more convenient for the CreditPortal provider (i.e., bank), as it does not always compute three separate and expensive customer evaluations, yet, RatingOne provides a more accurate customer
4.3. REASONING ON SERVICE BEHAVIOUR

evaluation. Anyway, the sub-service of CreditPortal which computes the customer rating can be replaced by either RatingOne or RatingTwo, not altering the behaviour of the whole CreditPortal application.

Let us now introduce another interesting issue in Service-Oriented Computing, such as the publication of correct service specifications. Indeed, generally, a service provider wants to publish a simpler specification (viz., an interface behaviour description) of its service by hiding unnecessary and/or confidential details of its implementation. The notion of behavioural equivalence introduced in Subsection 4.3.1 can be suitably employed to check whether the public specification properly advertises the internal behaviour of a service. In particular, given the OWL-S process models of a service and its public specification, we translate both into OCPR nets. Then we close the data places of the service that do not occur in the specification, and we check whether the resulting net is equivalent to the net corresponding to the specification.

For instance, Figure 4.25 depicts a possible public specification of the CreditPortal service (Figure 2.3), which hides several internal parameters and operations. Firstly, we translate both the full process model and the interface behaviour description of CreditPortal into OCPR nets. The resulting nets are illustrated in Figures 4.26 and 4.27, respectively. As one may note, all the data places of such two nets are open (i.e.,
they belong to the net interface). As a consequence, if we try to compare the net of Figure 4.26 with the net of Figure 4.27 with respect to the behavioural equivalence of Subsection 4.3.1, then we achieve a negative result, since the two nets have different
interfaces and they are hence externally distinguishable. The correct way to proceed – before equating the nets – is to close those data places that the provider does not want to be observed by the client. In the net of Figure 4.26 we would then close the following data places: securityEvaluation, rating, makeOffer and confirmation. Now, if we try to equate again the two nets, even if structurally different, they result to be externally indistinguishable. In other words, the simplified process model in Figure 4.25 is proven to be a correct interface behaviour description for the CreditPortal service.

So far, the issues of service replaceability and service publication have been addressed separately. It would instead be important to ensure that, after replacing in CreditPortal the component that evaluates customer reliability, the interface behaviour description illustrated in Figure 4.25 is still a correct abstraction of the new implementation of CreditPortal. Indeed, let us consider an OWL-S specification of CreditPortal where, e.g., the RatingOne service is plugged in. Clearly, the OCPR net obtained by its encoding (not depicted here) is not equivalent to the net obtained by the encoding of the original specification, as depicted in Figure 4.26. However, it clearly becomes so after closing with respect to the data places firstRating, secondRating and thirdRating. As a consequence, the external public OWL-S specification of CreditPortal presented in Figure 4.25 continues to properly describe the new CreditPortal implementation.

4.4 Related work on nets and net equivalences

The successful introduction of observational equivalences for process calculi in the early 1980s spawned researches on nets, as witnessed for instance by the survey [73]. According to the taxonomy there, saturated bisimilarity is a state-based equivalence, since it encompasses a notion of interface (a set of observable places) and it is dictated by the way the firing relation crosses the interfaces. Indeed, our bisimilarity is reminiscent of ST-equivalence, as in [73, Def. 4.2.6].

This Subsection does not aim at providing a survey of the field. First of all, the literature is very large, even if our main interest is in decidable compositional equivalences, thus restricting the area of possible intersection with former works.

In the rest of the Subsection we thus focus on two issues that are closely related to the novelties introduced in our framework: the use of the theory of reactive systems for obtaining a tractable equivalence, and the use of (equivalences on) nets for dealing with the specification of Web services and of their composition.

---

6The analysis there is restricted to contact-free C/E nets, i.e., such that the requirement \((M \cap t^\circ) \subseteq t^\circ\) of the enabling condition in Definition 4.4 holds for any reachable marking \(M\).
CHAPTER 4. BEHAVIOURAL ANALYSIS

Nets, open places and labels

Most recent formalisms for system specification come equipped with a reduction semantics: a suitable algebra of states is defined, and system evolution is represented by a relation between states. However, the lack of observable actions (either associated to the states, or to the reductions) inhibits the development of observational equivalences, which are often handier and more tractable.

Concerning Petri nets, the need of primitives for expressing the interaction with an environment was recognized early on, and notions of “net interface”, intended as a subset of the items of the net, are already reported in [73]. Interfaces are key ingredients for defining an observation, as well as for expressing net operators: along this line, a classical approach is the Box Algebra [14]. The main difference with our proposal is in the use of a set of labels for obtaining a labelled reduction relation, on top of which to define the weak semantics.

Quite related to our solution are open nets [6]: place/transitions nets where two distinguished sets of input and output places (where tokens may be added or removed, respectively) are identified, and then used to compose nets by coalescing places. The approach in [6] however differs from ours in that the authors stick to P/T nets and introduce a dichotomy between input and output places, which reflects on the type of the net composition. We refer to [6], and the references therein, for a survey and comparison with other interface-based techniques, mostly important the net components proposed by Kindler [40].

The most important source of inspiration for our work is the theory of reactive systems [43], introducing a technique for synthesising labels with suitable minimality requirements from a reduction relation that guarantees that the bisimilarity on the derived labelled relation is a congruence. Indeed, our approach benefits from the general definition of interface deriving from the theory. Concerning Petri nets, the technique was first applied by Milner in [56], after implementing nets into a more complex graphical structure, bigraphs, and later by Sassone and Sobociński [78], whose labelled relation largely coincides with ours. The main difference w.r.t. [78], besides the use of our flavour of C/E nets, is the introduction of saturated bisimilarity and the corresponding characterisation by weak bisimilarity. Moreover, our equivalence is weak (the number of transitions is not observed) and interleaving (parallelism is reduced to non-determinism). To the best of our knowledge, the treatment of such a bisimilarity for nets, and its decidable characterisation, is original to our work.

Nets for service equivalence

The application of Petri nets to the specification and the modelling of distributed systems has been around since their inception. Concerning Web services, the use of nets has been strongly advocated in the works by van der Aalst, see e.g. [85, 86] and the position paper [83]. More specifically, he and his coauthors address the issue of
4.4. RELATED WORK ON NETS AND NET EQUIVALENCES

equivalences for nets in [84]: they propose a trace-based, probabilistic equivalence for nets which is however quite far from our proposal.

In [50] WorkFlow nets [85] have been enriched (i.e., open WorkFlow nets) with communication places, which constitute the interface of the net. Yet, we introduced OCPR nets since both WorkFlow nets and open WorkFlow nets abstract from the (whole) data flow of a service, that, yet, is crucial for our discovery framework which searches, composes and analyses services relying on their ontology-annotated data.

Tightly related to us is the work of Martens, which actually inspired some of our examples in Section 4.3. More precisely, we refer to [47], where van der Aalst’s WorkFlow Petri nets (with disjoint sets of input, output and internal places) are analysed for checking structural properties of Web services. The author introduces there the notions of net module and net composition, which recall the open nets formalism mentioned above, and roughly coincide with our own solution.

Martens introduces in [47] the notion of net soundness and net usability, basically related to state reachability with respect to composition with suitable modules. We sketch here a solution for recasting those notions in terms of saturated bisimilarity. Let us start considering the OCPR net 1 depicted in Figure 4.28. Moreover, let \( \nu_O[-] \) be the OCPR context that close all the elements in the set \( O \) of open data places of an OCPR net \( N \). According to [47], we say that a net \( N \) is weakly sound if and only if \( \nu_O[N] \approx 1 \); a context \( C[-] \) utilizes a net \( N \) if \( \nu_O[C[N]] \approx 1 \); and a net \( N \) is usable if there exists a context \( C[-] \) using it.

![Figure 4.28: The OCPR net 1.](image)

Next, the author considers there a few notions of observational equivalence for his flavour of WorkFlow nets, discussing in turn trace, bisimulation and simulation equivalences. He identifies the weakness of trace equivalence with respect to deadlock, as we echoed in Section 4.3. He further argues on bisimulation, reaching the conclusion that simulation is the most adequate equivalence. Note that Martens’ notion of simulation (denoted here by \( \equiv \)) can be expressed by saturated bisimilarity. Indeed, \( N \equiv N' \) if and only if \( \nu_O[C[N]] \approx 1 \Leftrightarrow \nu_O[C[N']] \approx 1 \) for all possible contexts \( C[-] \): intuitively, this means that the two nets are usable by exactly the same environments. Note that our solution is stricter, since \( \approx \subseteq \equiv \), while the converse does not hold. Indeed, the nets \( M_3 \) and \( M_4 \) of Figure 4 in [48], now reproduced according to our notation in Figure 4.29, are equivalent according to Martens, even if they are not saturated bisimilar.
4.5 Discussion

In this Chapter we described the behavioural analysis of the discovery technique introduced in this thesis. The behavioural analysis first defines OCPR nets (for Open Consume-Produce-Read nets), to suitably model service behaviour. In particular, an OCPR net is equipped with two disjoint sets of places, namely, control and data places, to naturally model the control flow and the data flow of a service, and with an interface, which establishes those data places that can be observed externally. A formal encoding from OWL-S descriptions to OCPR nets is also presented. Then, the behavioural analysis introduces a suitable notion of behavioural equivalence for (OWL-S) services expressed as OCPR nets. The main features of such an equivalence are weakness, as it equates externally indistinguishable services by abstracting from the number of internal steps, compositionality, as it is also a congruence, and decidability, as the set of states that an OCPR net can reach is finite.

Given a set of services (e.g., previously selected by a functional analysis), the behavioural analysis first generates a composite service by performing the parallel composition of the OCPR nets modelling the behaviour of such services. Next, it provides two different behavioural analyses, according to the type of the client query to be satisfied. More precisely, it performs a lock analysis (to check whether a (composite) service terminates correctly after producing all the requested outputs) if a functional query is received, while it performs a bisimilarity analysis (to check whether a (composite) service features a requested behaviour) if a behavioural query is received.

It is worth observing that while by default the lock analysis is performed only when a functional query is received, it may be optionally executed also when a behavioural query is received, if requested by the client. The lock analysis indeed allows to detect (possibly) erroneous specifications in the behavioural client requests. For instance, the client may be wrong in requesting a particular service behaviour, which may be non-terminating. The approaches to service composition such as [34, 11, 12, 81, 72] aim at composing services so to solve a given composition goal,
yet, they do not check the real termination of the generated compositions.

While the OWL-S encoding in Subsection 4.1.4 provides a (formal) OCPR net representation of all the OWL-S control constructs as well as of the notions of composition, context and interface, we observe that:

- some OWL-S features, such as pre-conditions, effects and conditional outputs, are not considered. As we mentioned in Subsection 2.1.1, we decided to consider their possible employment nor within client requests neither within the discovery/composition process as their specification/usage is not quite clear. Yet, we note that conditional outputs may be translated into conditional constructs: for example, a BookStore atomic service with a conditional output book, may be translated into a (deterministic) choice between the BookStore with output book and an (additional) process representing a “book out-of-stock” failure.

- an OCPR net models the concept-flow of a service, rather than its actual data-flow. Indeed, we opted for a model stressing the persistency of data: once an OCPR data place has been inhabited, it cannot be emptied anymore. Our model thus abstracts away from data values.

The behavioural analysis is in charge of generating service compositions, however, only parallel composition is actually considered. Parallel composition is the “most general” composition in the sense that it allows services to freely interleave but for the constraints imposed by data dependencies. Indeed, for example, services may not be invoked sequentially, as considered in [61], since they may require to interact. For instance, consider a service $s_1$ which takes $B$ and produces $C$, and a service $s_2$ which first takes $A$ and produces $B$, and then takes $C$ and produces $D$. $s_1$ can be executed neither before nor after $s_2$. Yet, considering only parallel composition could be a limitation is some cases, and in particular, when a specific service behaviour is requested. For example, consider a behavioural request, asking for a service which takes $A$ and produces $B$, or takes $C$ and produces $D$. Suppose that two different services $s_1$, which inputs $A$ and produces $B$, and $s_2$, which inputs $C$ and produces $D$, are currently available. The behavioural analysis hence performs the parallel composition of $s_1$ and $s_2$. Such a composition, yet, does not satisfy the request, since it features a different behaviour (in particular, it does not implement the choice between executing $s_1$ or $s_2$). Other types of composition may be hence needed (e.g., as provided in [81, 11]) and we intend to investigate this issue in the future version of our discovery framework.

With respect to the state-of-the-art composition approaches, we observe that:

- our composition process engages a restricted number of services, suitably selected by the functional analysis. This is a crucial issue, since the high computational efforts usually required by service composition. Note, for example, that our discovery technique may be employed by the composition approaches
such as, e.g., [81, 11], to crucially reduce the number of services relevant for satisfying a composition goal.

- semantic information is employed to establish (data) dependencies among the services of the composition, conversely to [11, 12] which only take into account service behaviour.

- our composition approach considers a more flexible notion of service composition than [81, 72, 11, 12]. Indeed, it allows to match a service operation (e.g., taking as input $A$ and producing $B$) with a suitable interaction of several operations (e.g., a sequential composition of two operations, the first taking $A$ and producing $C$, and the second taking $C$ and producing $B$). The notion of service equivalence introduced in Subsection 4.3.1 then guarantees that the generated composition respects the given behavioural requirements.

Furthermore, it is also worth noting that the behavioural analysis:

- complements the functional analysis. When a functional request is received, the functional and behavioural analyses return those (compositions of) services which satisfy the client requirements and correctly terminate, thus advancing the approaches to a single service discovery (e.g., [44, 66, 79, 41, 7, 65, 1]) and the approaches to a composition-oriented discovery (e.g., [5, 8, 36, 61, 27]) which do not guarantee the correct interaction and termination of the returned services.

- provides a relevant contribution to service replaceability. The behavioural analysis employs the notion of service equivalence introduced in Subsection 4.3.1 (see Definition 4.18) to check whether two services are bisimilar, i.e., whether they feature the same externally observable behaviour. Let $s_1$ and $s_2$ two bisimilar services, then, $s_1$ can substitute $s_2$ in such a way that the change is transparent to the clients of $s_2$, and vice versa. Thus, when a behavioural request is received, every service (composition) outputted by the behavioural analysis can be safely used wherever the requested behaviour is needed.

- may be easily modified in order to cope with additional inputs. In Section 3.3, we indeed mentioned the possibility of extending the functional analysis in order to suggest additional inputs necessary to match otherwise unsatisfiable requests. In such a case, the behavioural analysis is still able to perform lock and/or bisimilarity analyses on the services selected by the functional analysis, yet using the suggested additional inputs to suitably expand the (functional or behavioural) requests.

Finally, we note that our relying on a notion of service bisimulation imposes more strict requirements than the context-specific replaceability relations in [69, 45, 74]. We also observe that our notion of replaceability emerges from an asynchronous
model (viz., Petri nets), and it is hence incomparable with notions of replaceability emerging from synchronous models, such as [28, 42]. Finally, as we noticed in Subsection 4.3.2, our relying on the concept of bisimilarity allows us to benefit from the wealth of tools and algorithms for checking system equivalence developed in the last decades.
Chapter 5

SAM: a proof-of-concept implementation

This Chapter is devoted to the presentation of the SAM prototype. SAM – for Service Aggregation Matchmaking – is a proof-of-concept implementation of the discovery framework previously described in Chapters 3 and 4. The global architecture of SAM is presented in Subsection 5.1, where we describe the behaviour of the SAM components and how they interact to feature the two main functionalities of the discovery framework, viz., the registration of a new service and the matching of client requests. The main implementation aspects (e.g., platform, language, libraries, etc.) are discussed in Section 5.2, while Section 5.3 illustrates the Web interface of SAM. Some experimental results and concluding remarks are presented in Sections 5.4 and 5.5, respectively.

Preliminary versions of the results presented in this Chapter have been recently published in [10, 25, 23].

5.1 An architecture overview

The global architecture of SAM is illustrated in Figure 5.1. SAM is a Web service designed to cope with two classes of users, clients and providers, which, mainly, can submit requests, and register new services, as reflected by the WSDL interfaces depicted in Figure 5.1. In particular, SAM supports service providers by featuring the following three operations:

1. createAccount(name, contactInformation) – to get a password for registering new services and ontologies,
2. submitOWL-Sservice(userPassword, serviceURI, serviceName, description) – to register a new (OWL-S) service,
3. submitOntology(userPassword, ontologyURI, ontologyName, description) – to register a new ontology.
A service provider first identifies itself by providing name and contact information to the `createAccount` operation, which assigns to the provider a password to access the submission service of SAM. Next, the provider is allowed to register new services and ontologies by suitably invoking the `submitOWL-Sservice` and the `submitOntology` operations, which both require provider credentials. The `submitOWL-Sservice` (submitOntology) operation inputs the URI, the name and the description of the service (ontology) to be registered, as well.

As illustrated in Figure 5.1, the authentication service of SAM is provided by the `ACCOUNTMANAGER` component, while the submission of new services and ontologies is supported by the suitable interaction of three components, viz., `OWL-S2PNML`, `ADDSERVICE` and `ADDONTOLOGY`. When a submission of a new ontology is received, the `ADDONTOLOGY` component – which implements the `AddOntology` function illustrated in Figure 3.4 of Subsection 3.1.2 – translates such an ontology into the hypergraph representation previously detailed in Subsection 3.1.1, where nodes and hyperedges respectively correspond to ontology concepts and relationships among them. In particular, `ADDONTOLOGY` exploits the external SemFiT component [39] (a service for “crossing” different ontologies) to establish equivalence and sub-concept relationships among ontology concepts.
Similarly, when a provider submits a new service to SAM, ADDSERVICE translates (the functional dependencies of) such a service into the hypergraph notation, while OWL-S2PNML models the service behaviour as OCPR nets. More precisely, the ADDSERVICE component implements the functions AddService, MakeProfile and Alt described in Figures 3.4, 3.2 and 3.3 of Subsection 3.1.2, respectively, and it exploits ADDONTOLOGY to translate the ontologies employed by the submitted services into the hypergraph representation. OWL-S2PNML [25] implements the encoding of OWL-S into OCPR nets introduced in Subsection 4.1.4, and, moreover, it generates the XML (viz., PNML [15]) code representing an OCPR net.

The internal representation of services (i.e., hyperedges and PNML documents) and ontologies (i.e., nodes and hyperedges) as well as account information are stored in the LOCALREGISTRY of SAM, which can be accessed by the SAM components by means of the REGISTRYMANAGER.

SAM allows clients to submit queries and to browse the local registry by providing the following four operations:

1. **functionalQuery**($inputs$, $outputs$) – to search for (compositions of) services that generate the requested outputs by taking as input (a sub-set of) the query inputs,

2. **behaviouralQuery**($processModel$) – to search for (compositions of) services that feature a query-equivalent behaviour,

3. **getOntologies**() – to get a list of the available ontologies,

4. **getServices**() – to get a list of the available services.

The REGISTRYBROWSER component provides the possibility of browsing the local registry, by listing the currently available services and ontologies.

The functional analysis, viz., the FunAn function described in Figure 3.13 (Subsection 3.2.1) is implemented by the FUNAN component, while the behavioural analysis consists of four (sub-)components, namely:

- **NETCOMPOSER** – which performs the parallel composition of a given set of OCPR nets (as summarised in Subsection 4.2),

- **OCPR2LTS** – which derives a labelled transition system from the firing semantics of a given OCPR net (as described in Figure 4.19 of Subsection 4.3.2),

- **LOCK ANALYSIS** – which checks whether a given service (viz., a labelled transition system) generates the requested outputs without locking (as summarised in Figure 4.20 of Subsection 4.3.2),

- **BISIMILARITY ANALYSIS** – which checks whether two services (viz., two labelled transition systems) feature the same (externally observable) behaviour (as described in Subsection 4.3.2).
When a functional client request is received, first the FUNAN component determines the sets of services capable of generating the requested outputs (viz., the outputs parameter of the functionalQuery operation) by taking as input (a sub-set of) the provided inputs (viz., the inputs parameter of the functionalQuery operation). Next, given a set of a services, NETCOMPOSER generates an OCPR net modelling the parallel composition of the services in such a set, and OCPR2LTS derives a labelled transition system from the firing semantics of the OCPR net. Then, the may(full)-termination of the labelled transition system is checked by the LOCK-ANALYSIS component. The functionalQuery operation hence returns those services (viz., the labelled transition systems) which pass the (may/must) termination check.

Similarly, when a behavioural client request\(^1\) is received, the components FUNAN, NETCOMPOSER and OCPR2LTS suitably cooperate by returning the labelled transition systems which model the (compositions of) services satisfying the functional requirements of the client request. Next, the BISIMILARITY ANALYSIS component checks whether a given (composite) service feature the same (externally observable) behaviour of the requested service (viz., the processModel parameter of the behaviouralQuery operation). Thus, the output of the behaviouralQuery operation consists of those service passing the bisimulation check.

It is worth observing that, due to efficiency reasons, the proof-of-concept implementation of the discovery framework orchestrates the functional and behavioural analyses in a generate-and-test pipeline. In such a way, the behavioural analysis can check each candidate set of services as soon as it is determined by the functional analysis, thus not waiting for all candidates to be determined and compared.

### 5.2 Some implementation details

SAM is deployed as a Java enterprise application. The Java Platform, Enterprise Edition 5 (Java EE, \url{http://java.sun.com/javaee/}) is the industry standard for developing portable, robust, scalable and secure server-side Java applications. Web services, component model, management, and communications APIs make Java EE suitable to implementing enterprise-class service-oriented architecture and next-generation Web applications.

The implementation of SAM has been conditioned by the following requisites:

- **Portability** – the system consists of Java packages, each of them wrapped in a standard Java EE component. Moreover, SAM is accessible as a Web service, described by a standard WSDL interface as well as by an OWL-S advertisement.

\(^1\)When an OWL-S process model expressing a behavioural query is received, it is analysed in order to determine the inputs and the outputs (viz., functional requirements), and the OCPR net modelling the behaviour (viz., behavioural requirements) of the desired service. For the sake of simplicity, we do not illustrate such an analysis in Figure 5.1.
5.2. SOME IMPLEMENTATION DETAILS

• **Extensibility** – SAM is deployed as a multitiered Java enterprise application, which allows for high levels of modularization and ease of substitute/add logic components (e.g., the integration with SemFiT, remotely accessed by its WSDL interface). Furthermore, the use of Java language allows us to employ many existing Java libraries and tools (e.g., OWL-S parsers).

• **Scalability** – Java EE platform natively guarantees scalability to component-based and multitiered applications.

• **Use of standards** – Besides the use of Java EE platform, the system implementation relies on other standard languages and well-known technologies:
  - PNML [15], to describe OCPR nets by means of standard XML files,
  - javaDB, to deploy the database (which is accessible via JDBC API directly by the Java EE component containers),
  - Mindswap OWL-S API, to validate, marshal/unmarshal OWL-S descriptions,
  - PNML framework API, to marshal/unmarshal PNML descriptions.

To ensure modularity and to keep distinct the implementation of SAM and the mechanisms of the Java EE platform, we wrapped the domain logic of SAM in three Java libraries: **SamFeedLogics**, which implements the submission of a new service or ontology (viz., the OWL-S2PNML [25], ADDSERVICE and ADDONTOLOGY components of Figure 5.1), **SamFunctionalLogics**, that implements the functional analysis (viz., the FUNAN component), and **SamBehaviouralLogics**, which implements the behavioural analysis (viz., the NETCOMPOSER, OCPR2LTS, LOCKANALYSIS, BISIMILARITYANALYSIS components). It is worth noting that each library is connected to the rest of the architecture by facade EJB (Enterprise Java Beans) components, that automatically retrieve other components’ references by Java EE server injection. Hence, each functional component is totally independent from the overall architecture, and it can be tested in a Java standard environment by employing suitable stubs and drivers. Furthermore, facade components declaratively instruct the application server (by means of Java 5 annotations included in the class files) to expose relevant methods as (WSDL) Web services.

The implementation of SAM is completed by the following Java EE components:

• **SamPersistence** – which, by abstracting from the actual data representation, provides two interfaces to respectively view (viz., DataReader of Figure 5.2) and modify (viz., DataWriter of Figure 5.2) – only upon authorization – the data contained in the local registry of SAM.

• **SamDBBrowser** – which implements a simple database browsing tool (viz., the REGISTRYBROWSER component).
• **SamAccountManager** – which grants (or denies) access to **SamFeedLogics** component. In particular, the current security management allows only registered users to submit new OWL-S descriptions and new ontologies. Moreover, it keeps trace of every submission to discourage any abuse. Future improvements of security management may be implemented in order to prevent possible leaks in quality of service.

• **SamGWTServlet** – which provides SAM with a friendly Web interface, developed with the Google Web Toolkit (http://code.google.com/webtoolkit/).

Figure 5.2 depicts the modular organization of the implementation of SAM. EIS (Enterprise Information System) tier represents the database management system (javaDB), while EJB (Enterprise Java Beans) tier and Web tier are two layers of the Java EE 5 server (Sun Java System Application Server 9), namely the EJB and the Web containers, respectively. Moreover, note that in Figure 5.2 white boxes represent Java EE components, while gray blocks represent the core SAM algorithms, each of them implemented as ordinary Java library. It is also worth observing that **SamFeedLogics** also includes the connection (viz., remote Web service invocations) with the **SemFiT** tool. Note, moreover, that for the sake of simplicity, Figure 5.2 shows a simplified WSDL SAM interface: in particular **query**, **browse**, **authenticate** and **submit** respectively denote the **functionalQuery** and **behaviouralQuery** operations, the **getOntologies** and **getServices** operations, the **createAccount** operation, and the **submitOWL-Sservice** and **submitOntology** operations, illustrated in Figure 5.1.

Further implementation details can be found in [9].

![Figure 5.2: SAM implementation.](image-url)

Finally, it is worth observing that the **SamBehaviouralLogics** library is still under development. Currently, only a prototypical implementation of the **OCPR2LTS** component is available. This is why in the next Subsection we restrict the discussion on
the Web interface and the actual behaviour of the SAM prototype to the functional part only.

5.3 SAM in action

SAM is available as a Web service, yet, we also provide a friendly Web interface to allow human clients to easily interact with SAM. In the following, we first describe the SAM Web interface (Subsection 5.3.1) as it appears when clients access it. Next, we illustrate how clients can easily submit (OWL-S) services (Subsection 5.3.2) and functional queries (Subsection 5.3.3) through the Web interface.

5.3.1 Web interface

When (human) clients access SAM, SAM provides the Web interface shown in Figure 5.3. As one can observe, the Web interface consists of four main parts:

- Top menu – which provides links to access other sections of the Web interface, such as the “list of services” section (which lists the services currently available
in the local registry of SAM), the “submit a service” section (which allows clients to submit an OWL-S service), and the “help”, “publications”, “authors” sections (which provide additional information on how to use the interface, and on the publications and authors of SAM).

- **Ontologies’ panel** – which shows a friendly tree-view of the ontologies currently available in the local registry of SAM. In particular, clients can browse such a tree to search and select ontology concepts for specifying their functional requests.

- **Query panel** – which, in turn, consists of the following three sub-parts:
  - **Query inputs’ panel**, which shows the ontology concepts selected as input of a functional client request. A “Add”/“Delete” button allows clients to add/remove concepts to/from the query inputs’ panel.
  - **Query outputs’ panel**, which shows the ontology concepts selected as output of a functional client request. A “Add”/“Delete” button allows clients to add/remove concepts to/from the query outputs’ panel.
  - **Submit query button**, which submit a (functional) query specifying as input the concepts in the query inputs’ panel and as output the concepts in the query outputs’ panel, to SAM.

- **Results’ panel** – which shows the (sets of) services selected by SAM to satisfy a given (functional) client request.

### 5.3.2 Submitting a service

The “submit a service” section (reachable from the top menu) of the Web interface allows providers to submit OWL-S services. As one can observe in Figure 5.4, the “submit a service” section presents several textual boxes which have to be filled with data concerning the service to be submitted. In particular, a provider wishing to submit a service, must provide its login and password to access the submission service of SAM, and the URI, name and description of the service to be submitted. Note that each box in Figure 5.4 corresponds to a parameter of the submitOWL-Sservice operation of the WSDL interface of SAM. Then, the provider sends its submission request to SAM by clicking the “submit service” button.

If the provider authentication succeeds, SAM retrieves the OWL-S service description available at the submitted URI, it generates the hypergraph and OCPR net representation of such description and adds them to its local registry, and finally, it informs the provider that the service submission succeeded. If some error occurs, the registry is not updated and a message error is returned to the provider.

For instance, Figure 5.4 illustrates the submission of the HotelService previously described in Section 2.1.2.
5.3.3 Submitting a functional query

When a client accesses the Web interface of SAM, the ontologies’ panel shows a tree-view of the ontologies currently available in the SAM local registry. Hence, a client can easily specify a functional query by browsing the ontologies’ tree and selecting the desired concepts, which will appear in the query inputs and query outputs panels of the Web interface. Then, the client submits its functional request to SAM by clicking the “submit query” button. Note that the lists of the inputs and the outputs in the query panel correspond to the parameters of the functionalQuery operation of the WSDL interface of SAM. The (sets of) services satisfying the functional request will appear in the Results’ panel of the Web interface.

For instance, Figure 5.5 illustrates the submission of a functional query specifying the following inputs and outputs:

- **inputs**: internationalConference, contactInformation, creditCard, hotel, and
- **outputs**: registrationReceipt, invoice.

Figure 5.6 illustrates the Web interface after submitting such a query. As one may observe in Figure 5.6, the Results’ panel shows the sets of services which functionally satisfy the submitted query. Note that, for each service, SAM specifies also the exact profile (viz., the exact sets of service operations) to be used. In particular,
Figure 5.5: Submitting a functional query to SAM.

the sample query can be fulfilled by two sets of services both consisting of (profiles of) the HotelService and ConferenceService.

In the current implementation of this proof-of-concept prototype, SAM denotes the profiles of services by internal IDs, which obviously are not useful for clients. Yet, we intend enhancing the results of SAM as the behavioural analysis implementation will be completed, so as to return a suitable specification of how to orchestrate the selected services in order to really satisfy the query.

5.4 Experimental evaluation

To evaluate the proof-of-concept implementation of SAM we employed the OWLS-TC v2 service retrieval test collection developed by the authors of OWLS-MX [41]. OWLS-MX, as previously described in Subsection 2.2.1, is a profile-based matchmaking system. The OWLS-TC collection consists of 576 OWL-S services covering several application domains (i.e., education, medical care, food, travel, communication, economy and weaponry). OWLS-TC also provides a set of 28 (functional) test

2Available at http://projects.semwebcentral.org/projects/owls-tc/.
A comparison between OWLS-MX and SAM is shown in Figure 5.7. For each OWLS-TC test query, we specify: (a) the number of services found by both the matchmakers (viz., OWLS-MX and SAM), (b) the number of services found by OWLS-MX only, (c) the number of single services found by SAM only, and (d) the number of compositions of services found by SAM only.

The experimental results in Figure 5.7 highlight that:

1. in some cases, OWLS-MX returns services not found by SAM,
2. generally, SAM finds more (compositions of) services than OWLS-MX.

Point (1) is due to the different notions of matching used by SAM and OWLS-MX. SAM states that a service satisfies a query if and only if every query output subsumes a service output, and every service input subsumes a query input. OWLS-MX, instead, states that a service satisfies a query if every query output either subsumes or is subsumed by a service output, and every service input either subsumes
or is subsumed by a query input. Yet, we argue that the notion of matching of OWLS-MX does not respect the OWL-S specification [64], which defines two processes type-compatible if for each output of one that flows to the input of the other, the type of the output is a subtype of the type of the input, that is, the output subsumes the input.

Concerning point (2), we observe that SAM simply performs a semantic (viz., ontology-based) matching, thus returning all those (sets of) services capable of producing the query outputs given the availability of the query inputs. OWLS-MX, instead, performs a hybrid matching which extends the semantic matching with a syntactic-based matching, also considering natural language text content of service descriptions in order to select services which belong to the same application domain of the given query. Yet, we argue that the OWLS-MX approach cannot be applied to a composition-oriented service discovery, where usually a query is satisfied by a set of several services of different domains (e.g., a travel agency may be composed of services of different domains such as transports, economy and geography).

On the other hand, OWLS-TC is a collection of atomic services and atomic test queries which can be satisfied by single atomic services. Indeed, it is worth observing that, while the OWLS-TC collection allows us to complete a first evaluation of SAM, it does not suffice to analyse all the potential (behavioural) features of SAM. To this end, more complex OWL-S service descriptions (i.e., non-atomic process models) are needed. We have then employed the SAM prototype to solve the example queries presented in Subsection 3.2.4: while SAM works properly and determines the compositions of services capable of satisfying the queries so as previously described in Subsection 3.2.4, we note that in these cases OWLS-MX would fail to find solutions.

Finally, we observe that searching for compositions of services obviously requires additional computational efforts. The experimental computations ran on Windows XP, Intel Pentium IV (3 GHz) with 1 GB RAM. Figure 5.8(a) illustrates the average response time of SAM, and in particular, it highlights the overhead due to considering compositions. However, as one may observe in Figure 5.8(b) such an overhead can be notably reduced if SAM is configured to perform semantic matching considering direct sub-concept relationships only.

5.5 Discussion

In this Chapter we presented a proof-of-concept prototype – named SAM, for Service Aggregation Matchmaking – which implements the discovery framework previously described in Chapters 3 and 4.

The current implementation of SAM is capable of processing functional queries, that is, queries which specify the (ontology-annotated) inputs and outputs of the service to be found. As described in Section 5.3, SAM provides a friendly interface, which allows (human) clients to browse the available ontologies, so to easily formulate functional queries. When a functional query is received, SAM returns the sets of
5.5. DISCUSSION

Figure 5.7: A comparison between SAM and OWLS-MX.

<table>
<thead>
<tr>
<th>Query</th>
<th>Numbers of services found by OWLS-MX only</th>
<th>SAM only (single services)</th>
<th>SAM only (compositions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
<td>15</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>17</td>
<td>55</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>17</td>
<td>39</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>29</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>21</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>12</td>
<td>5</td>
<td>–</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>–</td>
<td>37</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>–</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>–</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>5</td>
<td>–</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>17</td>
<td>10</td>
<td>2</td>
<td>–</td>
</tr>
<tr>
<td>18</td>
<td>11</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>19</td>
<td>12</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>12</td>
<td>–</td>
<td>4</td>
</tr>
<tr>
<td>21</td>
<td>12</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>22</td>
<td>13</td>
<td>–</td>
<td>3</td>
</tr>
<tr>
<td>23</td>
<td>13</td>
<td>–</td>
<td>12</td>
</tr>
<tr>
<td>24</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>11</td>
<td>–</td>
<td>14</td>
</tr>
<tr>
<td>26</td>
<td>14</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>27</td>
<td>15</td>
<td>–</td>
<td>3</td>
</tr>
<tr>
<td>28</td>
<td>17</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

Figure 5.8: Average query response time of SAM.

(a) single vs composition-oriented service discovery

(b) exact match vs flexible match

services capable of satisfying the given query. Furthermore, SAM allows service providers to submit (OWL-S described) services. When a service submission request is received, SAM translates the service into the internal representation and store it
into the local repository.

Yet, while the implementation of the functional analysis of SAM has been completed, the implementation of the behavioural analysis is still under development. In particular, only a prototypical implementation of the OCPR2LTS component is currently available. Our own plans for future work hence include:

(1) to complete the implementation of the behavioural analysis,

(2) to extend the interface in order to allow clients to submit behavioural queries,

(3) to enhance the “answers” of SAM, for instance, by providing a suitable specification (e.g., an OWL-S process model) of how to orchestrate the selected services in order to satisfy the solved query, and

(4) to experiment SAM on a large number of (functional and behavioural) queries and (non-atomic) service descriptions.
Chapter 6

Concluding remarks

In this Chapter we summarise the main contributions of the thesis in Section 6.1, and in Section 6.2 we draw some final concluding remarks on the applicability in practice of the proposed techniques.

6.1 Summary

In this thesis we have defined a suitable technique for a composition-oriented service discovery which exploits both ontology and behaviour information available in descriptions of services. Such a technique contributes to three important open challenges of the service-oriented computing paradigm, namely:

- *service discovery*, since it is capable of reasoning on services’ capabilities, thus not failing to locate services due to syntactic mismatches,
- *service composition*, since it is capable of generating (non-locking) service compositions, when no service satisfies a given request by itself,
- *service replaceability*, since it is capable of finding (compositions of) services behaviourally equivalent to a desired service.

Currently, the discovery, the composition and the replacement of services are hampered by the following two main issues:

1. the *lack of ontology* and *behaviour* information in standard WSDL service descriptions, which inhibits the possibility to reason on service capabilities, to properly compose services, and to *a priori* check service properties such as lock-freedom.

2. the availability of *heterogeneous* service descriptions, which inhibits the possibility to discover services, to compose them as well as to check their interaction protocol in an automatic way.
In order to tackle limitation (1), we argue that providers should advertise their services with semantics- and behaviour-enhanced descriptions, as we will better discuss in Subsection 6.2.2. In particular, in this thesis we considered services exposing OWL-S descriptions. OWL-S [64] is a computer-interpretable semantic mark-up language, which provides an ontology-based description of service capabilities, and a declaration of the interaction service behaviour.

Furthermore, to tackle the issue of service heterogeneity, we defined a suitable common formalism to represent (ontology and behaviour) information available in heterogeneous service descriptions. Such a formalism consists of:

- a **dependency hypergraph** — to represent semantics relationships among ontology concepts, and data-flow dependencies among services. Intuitively, such an hypergraph consists of nodes, which correspond to ontology concepts, and hyperedges, which represent dependencies among them. More precisely, a hyperedge can represent: (1) a *subConcept relationship*, which links a concept with its sub-concepts, (2) an *equivalentConcept relationship*, which links together two (semantically) equivalent concepts, and (3) an *intra-service dependency*, which connects the inputs and the outputs of a particular service behaviour.

- **OCPR nets** — to model behaviour of services. Briefly, an OCPR net (for Open Consume-Produce-Read net) is equipped with two disjoint sets of places, namely, *control* and *data* places, to naturally model the control flow and the data flow of a Web service, and with an *interface*, which establishes those data places that can be observed externally.

The discovery technique presented in this thesis consists of two main elements:

1. a **functional analysis**, to select services, and
2. a **behavioural analysis**, to check service properties and services' equivalence.

The **functional analysis** of the proposed discovery technique provides:

- A suitable *translation mechanism* to generate the hypergraph representation of OWL-S service descriptions and of the ontologies used by such descriptions.

- A sound and complete *discovery algorithm*, which, given a client request specifying inputs and outputs of the desired service, explores the dependency hypergraph in order to (automatically) determine (minimal sets of) services capable of satisfying the request. Briefly, a set of services satisfies (the functional requirements of) a client request if such services generate the requested outputs, and if they take as input (a sub-set of) the inputs of the client request.

The main features of the functional analysis are:
(a) *Ontology-based matching* — the functional analysis is capable of performing flexible matching between service inputs and outputs automatically, so to establish functional dependencies among services. In particular, an output of a service is matched with an input of another service if the service output is described by an ontology concept equivalent to or more specific than the ontology concept which describes the service input.

(b) *Composition-oriented matching* — the functional analysis is able to generate suitable service compositions (possibly including multiple instances of services), so to satisfy queries which can not be fulfilled by a single service.

(c) *Minimal sets of services* — the functional analysis returns only minimal sets of services, namely, sets only containing services strictly necessary to satisfy a given request.

The *behavioural analysis* of the proposed discovery technique provides:

- A formal, compositional *encoding* from OWL-S descriptions to OCPR nets.
- A *lock analysis* to check whether a (composite) service satisfies a given request (viz., it generates the requested outputs) without locking.
- A *bisimilarity analysis* to checks whether a (composite) service features a desired behaviour. In this setting, a suitable notion of *behavioural equivalence* for Web services expressed as OCPR nets is defined. Three properties of such an equivalence are *weakness*, as it equates externally indistinguishable services by abstracting from the number of internal steps, *compositionality*, as it is also a congruence, and *computability*, as the set of states that an OCPR net can reach is finite. In particular, compositionality permits to exploit the equivalence relation for a disciplined incremental development of services, by means of sound compositions and replacements.

Functional and behavioural analysis suitably interact in order to provide a *discovery framework* offering two main functionalities, that is:

1. *to register new services* — when a provider submits a (new) OWL-S service to the framework, it (i.e., the functional and behavioural analyses, respectively) parses the OWL-S service description in order to generate a (portion of the) hypergraph as well as an OCPR net which model the new service, and it adds them to a local service registry.

2. *to search for services* — the framework is capable to tackle two types of queries:
   - *functional queries* — that is, queries specifying the functional attributes (viz., inputs and outputs) of the desired services. In such a case, the framework (in particular, the functional analysis and the lock analysis) determines the (compositions of) services which satisfy the query without locking.
• behavioural queries – that is, queries specifying the functional attributes and the expected behaviour of the desired services. In such a case, the framework (in particular, the functional analysis and the bisimilarity analysis) determines the (compositions of) services which satisfy the query by featuring the desired behaviour.

In this thesis we also presented a proof-of-concept prototype – named SAM, for Service Aggregation Matchmaking – which implements the main functionalities of the described service discovery framework.

While we believe that the results presented in this thesis represent an advancement in the state of the art in the discovery, composition and replaceability of services, further investigation is obviously needed in these areas. For instance, our own plans for future work include:

(1) To consider data multiplicity. Indeed, the presented discovery technique abstracts from multiplicity of data in modelling services (e.g., it abstracts from how many instances of service inputs and outputs are really required and produced).

(2) To consider other notions of service equivalence. We introduced a behavioural congruence for Web services based a notion of service bisimulation, yet, more relaxed notion of service equivalence may be used.

(3) To engineer the proof-of-concept implementation and to experiment it on a large number of queries and service descriptions.

(4) To develop a suitable technique to index services in order to feature a scalable discovery technique applicable on large service registries.

(5) To extend the discovery framework to deal with heterogeneous (i.e., not necessarily OWL-S) service descriptions.

In the following Section, we discuss points (4) and (5), since they are crucial for the applicability in practice of the discovery technique presented in this thesis.

6.2 On the applicability of the proposed techniques

In this Section we discuss several issues related to the applicability in practice of the discovery technique presented in Chapters 3 and 4. We discuss scalability issues in Subsection 6.2.1 and some approaches to deal with heterogeneous (i.e., not necessarily OWL-S) service descriptions in Subsection 6.2.2.
6.2. ON THE APPLICABILITY OF THE PROPOSED TECHNIQUES

6.2.1 On the scalability of service discovery

In Chapters 3 and 4 we have presented a suitable technique for a behaviour- and ontology-aware service discovery, capable of finding (compositions of) services behaviourally equivalent to a desired service. Functional and behavioural analyses of service descriptions can guarantee high quality discovery results, however, they are time consuming tasks. A scalability issue hence emerges when such a discovery technique is to be applied on a large service registry. More efficient algorithms for functional and behavioural analyses may be developed, yet, we argue that possible improvements of their time complexity do not suffice to address the scalability of the discovery technique. Moreover, we observe that also the (obvious) possibility of caching queries’ results does not suffice to guarantee scalability. Indeed, an efficient search mechanism would be needed to find cached service compositions (e.g., to rapidly select cached compositions providing more outputs than the query and requiring less input than the query, or providing equivalent/sub concepts of the query outputs, and so on). Instead, a really promising solution is indexing services. Obviously, indexing services is more complex than indexing text, since a service description provides different types of information, such as an ontology-annotated description of service inputs and outputs, as well as a declaration of the interaction protocol of services.

In the rest of this Section, we first briefly describe well-known techniques for indexing large document collections, next, we discuss how such techniques might be employed to indexing Web services. It is worth observing that the scope of this Section is not to propose a solution to indexing services, but rather to discuss some of the main issues toward the development of suitable techniques for indexing services.

Indexing documents

When a large collection of documents is available, a major problem is to find documents which contain a requested word (or phrase). A naive approach would be to sequentially scan each document to search for the given word. Yet, an obvious flaw of such an approach is that it does not scale to large document collections. A common solution is to pre-elaborate the documents in order to convert them into a format that allows to search for words rapidly, hence not scanning the entire collection of documents. Such a conversion process is called indexing, and its output is called index. Roughly, an index is a data structure that allows fast (random) access to words stored inside it. For instance, the index at the end of a book allows us to quickly locate pages containing a specific word.

Inverted lists [35] are currently the most used indexing technique for large document collections. The two main components of inverted lists are:

- Dictionary (or lexicon) – which is the set of distinct words contained in a document collection,
• Posting lists – which, for each word \( w \), contain all the occurrences (viz., pairs \((\text{documentID}, \text{word position})\)) of \( w \) within the collection.

For example, let us consider the toy collection depicted in the left part of Figure 6.1. Such a collection consists of four documents, viz., documents 01, 02, 03 and 04, each of them containing some text (viz., a list of words). For instance, the document 02 contains the text \( b b e a e a d c \). Note that the table in the left part of Figure 6.1, which associates each documentID with the text the document contains, is also named forward index. Suppose that the dictionary of the collection consists of five words, viz., \( a, b, c, d \) and \( e \). Then, to index the collection we construct the posting list for each word of the dictionary. For example, the posting list of the word \( d \) is \((02,7),(03,5)\), since \( d \) occurs in document 02 in position 7 (viz., \( d \) is the seventh word of the document 02) as well as in document 03 in position 5 (viz., \( d \) is the fifth word of the document 03). The right part of Figure 6.1 depicts the so called inverted index, which can be achieved by reversing the forward index, and which associates each word of the dictionary with the corresponding posting list.

\[
\begin{array}{|c|c|c|}
\hline
\text{docID} & \text{Text} & \text{Dictionary} \\
\hline
01 & ace & a \\
02 & b b e a e a d c & b \\
03 & c c e e d & c \\
04 & a b c & d \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{Dictionary} & \text{Posting lists} \\
\hline
a & (01,1),(02,4),(02,6),(04,1) \\
\hline
b & (02,1),(02,2),(04,2) \\
\hline
c & (01,2),(02,8),(03,1),(03,2),(04,3) \\
\hline
d & (02,7),(03,5) \\
\hline
e & (01,3),(02,3),(02,5),(03,3),(03,4) \\
\hline
\end{array}
\]

Figure 6.1: Indexing textual documents.

Inverted lists are one of the key ingredients of the most widely known search engines. For example, let us roughly overview the anatomy of Web search engines [20]. Three major parts can be identified:

• Crawling – that is, the downloading of Web pages. Several distributed crawlers (or spiders) download Web pages and store them into a repository. Each Web page has an associated ID number, called docId.

• Indexing – that is, the construction of an (inverted) index of Web pages. The indexer reads documents in the repository, parses them, and converts each document into a set of word occurrences in order to generate a forward index. Then, the forward index is reversed into an inverted index, thus suitably generating the dictionary and the posting lists.
6.2. ON THE APPLICABILITY OF THE PROPOSED TECHNIQUES

• Searching – that is, the (rapid) location of Web pages satisfying client requests. When a client request (viz., a list of words) is received, the searcher accesses the inverted index to retrieve the posting lists of the requested words. Then, it returns the documents (viz., Web pages) which results from the intersection of such posting lists. For example, let us consider the toy document collection in Figure 6.1. If the searcher receives a query for the words a, c and e, it returns the documents 01 and 02.

We now briefly discuss whether inverted lists-based indexing techniques can be employed to index Web services providing semantically and behaviourally enhanced service descriptions.

Indexing Web services

In the following, we first discuss how to generate inverted index of non-annotated services. Second, we include service compositions into the inverted index, and in particular we consider the possibility of indexing either all the possible compositions of the available services, or the compositions satisfying all the possible client requests. Finally, we take into account the indexing of semantically annotated services.

Generating inverted index of non-annotated services.

Each Web service is advertised by a service description, which provides (at least) a list of the service capabilities (viz., service inputs and outputs). Let us now consider a service description as a document, and service capabilities as words occurring inside it. In this setting, it is possible to employ inverted lists to index (descriptions of) services. Indeed, service descriptions collected into a service repository constitute the document collection to be indexed, while service inputs and outputs employed by such descriptions compose the dictionary of the collection. Then, each word (viz., each service capability) of the dictionary should be associated with the list of documents (viz., service descriptions) where such a word occurs.

Yet, as stated in Definition 3.7 (Section 3.2), a service satisfies a (functional) query if it produces all the requested outputs by taking as input (a subset of) the query inputs. Namely, in order to match a given (functional) query, a service description (viz., a document) must contain all the query outputs, while it may contain the query inputs. To be more exact, all the service inputs occurring in the service description must be contained in the query inputs. As a consequence, we can associate a service capability with the list of those service descriptions where such a capability occurs as an output. Hence, to satisfy a client query specifying the inputs and outputs of the desired service, we can first perform the intersection of the lists of services associated to each query output. Then, for each of the selected services we can check whether its inputs are contained in the query inputs. If so, such a service functionally satisfies the given query.
Generating inverted index of compositions of *non*-annotated services.

We observe that associating service capabilities with *single* services (where they occur as output) does not suffice in case of complex client requests which need to compose the capabilities of different services. Indeed, if there exists no service capable of satisfying the query by itself, it would be necessary to execute the discovery algorithm (see Subsection 3.2.1), thus not taking real advantage of the constructed index. A brute-force solution is to pre-compute all the possible compositions of the services in the repository, so to associate service capabilities directly to service compositions (where they occur as outputs).

Let us consider, for instance, the sample service repository in Figure 6.2. The repository contains three services, namely, S01 (which inputs c1 and returns c2 and c3), S02 (which inputs c4 and produces c3 and c5) and S03 (which takes as input c6 and returns c7). First, we (pre-)compute all the possible compositions of S01, S02 and S03, and we assign a composition number ID (viz., complID) to each of them, as illustrated in the top part of Figure 6.2. For example, the complID C06 denotes the composition of the services S01 and S03. Then, we construct the inverted index, by associating each service capability of the dictionary (viz., c1, c2, c3, c4, c5, c6, and c7) to those compositions where such a capability occurs as an output, as illustrated...
6.2. ON THE APPLICABILITY OF THE PROPOSED TECHNIQUES

In the right part of Figure 6.2. For instance, the capability c2 can be produced by the compositions C01, C04, C06 and C07, while capability c7 can be produced by the compositions C03, C05, C06 and C07. Let us now consider for example the query asking for a service which takes as input c1 and c4 and produces as output c2, c3 and c5. First, we perform the intersection of the lists of services associated to the capabilities c2, c3 and c5, hence selecting compositions C04 and C07. Then, we check whether the inputs of composition C04 as well as of composition C07 are contained in the query inputs. As one may observe in the top part of Figure 6.2, this holds only for composition C04 which hence is capable of satisfying the request.

Note that the composition number IDs can be exploited to keep ordered the posting lists of the inverted index, so to speed up the computation of the posting lists' intersection.

Several issues yet emerge from the rough indexing solution described so far. A first problem is how to generate service compositions. As discussed in the above example, a possibility is to generate all the possible compositions of the services available in the repository. Yet, such a solution may produce a lot of useless compositions which increase the index dimension without enhancing the searching process. For instance, if the repository contains thousands of services, probably it has no sense generating all the compositions of such services, and in particular, those compositions which consist of thousands of possibly non related services (e.g., service operating in totally different domains).

Another possibility to pre-compute service compositions may be generating all the possible requests and consequently determining the compositions capable of satisfying them. Obviously, also in this case it is not reasonable to generate all the possible combinations of the capabilities of the available services, yet, suitable heuristics may be employed to avoid of generating useless requests. For instance, the heuristics may take advantage of the previous client requests to determine the most requested service capabilities, to identify the service capabilities usually requested together, to estimate the (maximum) number of services which may be fruitfully composed together, and so on.

Figure 6.3 illustrates a partial sketch of a possible revised architecture of the discovery framework presented in this thesis in order to integrate the indexing of services. As one may observe in the figure, functional requests are addressed by the SEARCHER component, which exploits the index to rapidly satisfy the requests. Similarly, the behavioural requests are first processed by the SEARCHER, which selects the sets of services which functionally satisfy the request. Then, the bisimilarity analysis checks whether such services feature the same (externally observable) behaviour of the request. A QUERY SERVER component generates possible requests, by taking into account the DICTIONARY (viz., the service capabilities currently available) and the submitted client requests previously addressed by the discovery framework. A battery of crawlers, each of them executing the FunAn function described in Figure 3.13 (in Subsection 3.2.1), solves the queries generated by the
CHAPTER 6. CONCLUDING REMARKS

QUERY SERVER. The results are analysed by the LOCK ANALYSIS, which stores only the non-locking service compositions (viz., the forward index, which associates each composition to its inputs and outputs) into the INDEX. Each FUNAN component accesses the local service registry by means of the REGISTRY MANAGER interface. Finally, the INDEXER reverses the forward index to generate the inverted index, which associate each service capability to the compositions producing as output such a capability and which updates the DICTIONARY of the service collections.

Figure 6.3: A minimalist architecture for indexing services.

Generating inverted index of semantically-annotated services.
Let us now take into account how to index compositions of semantically-annotated services. Several issues emerge. Indeed, when there is no composition capable of producing all the query outputs, there may exist a composition producing sub-concepts and/or equivalent concepts of such query outputs. For example, let us consider a query specifying price and DVD as service outputs, and an inverted index associating the concepts price, cost, DVD and musicalDVD to the compositions $C_1$, $C_3$, $C_2$ and $C_3$, respectively. As one may observe, there is no composition which produces the requested outputs price and DVD, yet, composition $C_3$ can produce cost and musicalDVD, which are an equivalent concept and a sub-concept of price and DVD, respectively. A possible solution to cope with the indexing of semantically enhanced service descriptions is query expansion. Before processing a query, it is expanded (viz., various queries are generated) into all the possible combinations of the query parameters and their sub-/equivalent concepts. Then, each single query is processed separately. For example, the previous query specifying price and DVD as service outputs can be expanded into four queries, each of them specifying as outputs: (1) price and DVD, (2) cost and DVD, (3) price and musicalDVD and (4) cost and musicalDVD, respectively. In particular, the (fourth) query can be satisfied.
by composition $C_3$. However, in case of complex requests, a large number of queries may be generated, thus not speeding up the query answering task.

Inverted lists cannot be hence directly employed to index semantically-annotated services, yet we mention two possible directions for future investigation:

(1) to suitably extend inverted lists to store advanced information (viz., enriched inverted index), and

(2) to develop ontology-aware mechanisms to suitably search (semantic) information in the inverted index.

Finally, it is also worth to discuss the role of behavioural information in indexing services. As described in Chapters 3 and 4, functional information is employed to select services, while behavioural information is employed to compose and analyse them. In particular, service behaviour is analysed to establish whether two different services feature an equivalent (externally observable) behaviour. Then, it is reasonable that “clusters” of behaviourally equivalent services may be created, so that if a service $s_1$ is equivalent to a service $s_2$, $s_1$ is equivalent to each of the services which belong to the cluster of $s_2$, as well.

6.2.2 Dealing with other service description languages

In Chapters 3 and 4 we presented a composition-oriented discovery technique capable of exploiting semantics and behaviour information available in OWL-S service descriptions. Yet, Web services are mostly described by means of WSDL interfaces, while a small part of them is described with a variety of recently proposed service description languages, such as WS-BPEL [19], SAWSDL [30], WSMO [94], and OWL-S [64]. Hence, the heterogeneity in Web services concerns both the description language employed to publish them, and, more importantly, the type of the exposed information. It is worth to observe that (“non-OWL-S”) service descriptions providing semantics and behaviour information (e.g., WSMO) can be tackled by the discovery technique presented in this thesis, if a suitable translator of such a language into the internal service representation (viz., dependency hypergraph and OCPR nets) is available.

In this Subsection we (roughly) discuss how the discovery technique in Chapters 3 and 4 can deal with service description languages providing behaviour information only (e.g., WS-BPEL), and with service description languages providing neither semantics nor behaviour information (e.g., WSDL). Finally, we also discuss how such a discovery technique can be embedded into existing UDDI registries in order to deal only with semantics- and behaviour-annotated services, thus leaving UDDI to cope with WSDL services.
Taking into account WS-BPEL processes

WS-BPEL has been recently approved as a new OASIS standard for expressing Web service compositions. Basically, a WS-BPEL process orchestrates the operations offered by the partner Web services through WSDL interfaces. A WS-BPEL process defines the behaviour of a service composition by relating a number of basic (i.e., invoke, receive, reply, wait, assign, throw and empty) or structured activities (i.e., sequence, switch, pick, flow, compensate, scope and while).

With respect to behaviour information, the discovery framework presented in this thesis can easily cope with WS-BPEL processes, for example:

(1) By using a BPEL2OWL-S translator. The availability of a BPEL2OWL-S translator [4] would avoid the need of developing specific algorithms to generate the OCPR net representation of WS-BPEL processes. Yet, a limitation of [4] is that BPEL2OWL-S does not cope with all the WS-BPEL structured activities.

(2) By adapting existing BPEL2PN translators. In the literature there are several proposals to expressing BPEL processes by means of Petri nets, e.g., [62, 77, 86] introduce a mapping from WS-BPEL to Petri nets, high-level Petri nets, and WorkFlow nets, respectively. Tools for translating WS-BPEL to Petri nets are also available (e.g., [63, 38]). We can hence benefit from the state-of-the-art to easily define a mapping from WS-BPEL to OCPR nets in the style of the compositional encoding from OWL-S to OCPR nets formalised in Subsection 4.1.4.

The main issue toward a full integration of WS-BPEL processes into the discovery framework presented in this thesis is however the lack of semantics information in WS-BPEL processes. As a consequence, the hypergraph representation of a WS-BPEL process cannot be generated, thus inhibiting the possibility of our discovery technique (in particular, the functional analysis) to cope with WS-BPEL processes. Hence, with respect to semantics information, two possible approaches to cope with WS-BPEL processes are:

(1) to automatically infer ontology annotations from WSDL types,

(2) to manually annotate WS-BPEL processes.

In order to infer ontology annotations automatically, we distinguish two approaches, namely to infer annotations

(a) before inserting a WS-BPEL process into the framework, or

(b) after inserting a WS-BPEL process into the framework.
In the case (b), the discovery framework considers the names of (WSDL) parameters (viz., the parts of the input/output messages exchanged by the process) as ontology concepts. For instance, we can employ the BPEL2OWL-S [4] translator to perform a syntactic transformations of WS-BPEL processes into OWL-S descriptions. We can hence define an ontology including all the parameters of the WS-BPEL process, then exploiting ontology crossers (e.g., the SemFiT tool, Subsection 3.1.1) to determine semantic relationships with other ontologies. In such a way, we can construct an algorithm suitably generating the hypergraph representation of WS-BPEL processes, in the style of the AddService function (Figure 3.4, Subsection 3.1.2) which copes with OWL-S services. Obviously the quality of the results of the ontology crossers strongly depends on the names of parameters chosen by the provider. Indeed, a FlightDepartureDate parameter may be easily related with other ontology concepts by an ontology crosser, yet, a FDD parameter may be related to a floppy disk driver. Furthermore, this solution promotes the fragmentation of ontologies, in contrast with the Semantic Web vision that aims at describing available resources with a set of global, shared ontologies.

In the case (a), (semi-)automatic ontology annotators (e.g., [37, 54]) may be used to give semantics to WS-BPEL processes. Ontology annotators implement machine learning techniques to infer the meaning of service parameters from the analysis of their names and textual descriptions. Each parameter is hence automatically annotated (if possible) with a concept of some existing ontology. Yet, the success of the ontology annotators, and then of the ontology crossers still depends on the suitability of the parameters’ names and descriptions.

In order to manually annotate WS-BPEL processes, two approaches can be mentioned:

(a) to ask providers to submit – together with the WS-BPEL process – a suitable (manually defined) mapping, which associates each process parameter to a concept defined in some available ontology,

(b) to require semantically enhanced WS-BPEL processes. In this setting, it is worth mentioning the work of Pistore et al. that propose in [72] a minimalist approach to semantically annotate WS-BPEL processes. Briefly, they extend WS-BPEL specifications by introducing the new attribute “semann” which (mainly) associates the (parts of the) input and output messages of the process with ontology concepts.

Semantics annotations of WS-BPEL specifications allow us to suitably generate the hypergraph representation of WS-BPEL processes, by first inserting into SemFiT and into the hypergraph (if not already present) the ontologies employed by the annotations and then adding to the hypergraph the functional dependencies within WS-BPEL processes. The solution which requires semantically enhanced WS-BPEL processes is hence preferable to the solution which automatically gives
semantics to WS-BPEL processes, since the latter solution often leads to low quality results (e.g., a WS-BPEL process may be represented into the hypergraph by isolated dependencies not linked to the other services available).

Taking into account WSDL services

The problem emerging from coping with WSDL service descriptions is twofolded, since they provide neither semantics nor behaviour information. With respect to semantics information, two possible approaches to cope with WSDL descriptions are:

1. To automatically infer ontology annotations from WSDL types. For instance, we can syntactically translate the (parts of the) input and output messages of WSDL descriptions into (isolated) ontology concepts (we can also use automatic ontology annotators like [37, 54], and there are syntactic-based translators available like [67]) and then employ suitable ontology crossers (e.g., the SemFiT tool) to relate such new concepts to the available ontologies.

2. To manually annotate WSDL descriptions. Alternatively, a manual approach consists of enabling the submission of only semantically enhanced WSDL descriptions (e.g., SAWSDL descriptions [30], or WSDL descriptions complemented by a (manually provided) mapping from WSDL messages to concepts defined in some available ontologies).

The hypergraph representation of (automatically/manually) annotated WSDL descriptions can be suitably generated and, in particular, a functional (viz., intra-service) dependency is inserted to the hypergraph for each WSDL operation.

Yet, while there are approaches available to add semantics to WSDL, there is no way to determine the behaviour of a WSDL described service. Hence, with respect to behaviour information, two possible solutions to deal with WSDL descriptions are:

1. to infer a simple behaviour from WSDL interfaces, and

2. to require service behaviour.

In case (2), providers should be required to submit – together with the WSDL description – a suitable specification (e.g., a Petri net) of the service behaviour. In such a case, the OCPR net representation of a WSDL service would be easily generated. Yet, it is reasonable to imagine that a service provider wishing to describe the behaviour of its service would probably uses WS-BPEL specifications.

In case (1), a WSDL service can be associated with the only possible behaviour which can be deduced from its description, namely, all the operations that it provides are always invocable. In such a case, the OCPR net representation of a WSDL service consists of a choice among the operations provided by the service.
6.2. ON THE APPLICABILITY OF THE PROPOSED TECHNIQUES

However, assuming such a general behaviour of (WSDL) services affects the reliability of the discovery results of the framework presented in this thesis. Indeed, a composition which includes one or more WSDL services cannot be guaranteed to satisfy a client request without locking, since the behaviour of such WSDL services is unknown. In particular, executing a behavioural analysis on a set of services associated with a (possibly) wrong behaviour is useless. This suggests to limit the matching of WSDL services to the analysis of their semantically annotated functional attributes (i.e., input/output messages). Hence, a possible solution to integrate WSDL services into the discovery framework presented in this thesis consists of (1) generating the hypergraph representation of the semantically annotated WSDL services, and (2) executing the functional analysis and directly returning to the client those selected sets of services which include (at least) a WSDL service, not executing the behavioural analysis. It is worth noting that – in case of (semantically annotated) WSDL services – the quality of the results achieved by the discovery framework of this thesis is similar\(^1\) to the quality of the results achieved by those approaches performing a semantics-based (composition-oriented) matching such as [5].

### Considering only ontology- and behaviour-annotated services

In this Subsection we would also like to mention an architectural choice alternative to the one presented in Section 5.1, that is, to embed the discovery framework of this thesis in UDDI registries, similarly to the approach of Srinivasan et al. [79]. In such a way, the discovery framework deals only with ontology- and behaviour-annotated services (e.g., OWL-S) and addresses functional and/or behavioural queries, while UDDI copes with the standard WSDL services and addresses syntactic queries.

This scenario is sketched in Figure 6.4. A first prerequisite of such an approach is that the discovery framework should have access to semantics and behaviour information provided by (OWL-S or semantically enhanced WS-BPEL) service descriptions. Hence, semantics and behaviour information need to be embedded inside UDDI advertisements, for instance, by providing a suitable mapping from OWL-S/WS-BPEL elements into specialised UDDI tModels, in the style of the UDDI mapping mechanisms employed in [79, 87]. As in [79], UDDI registries should provide two special ports (viz., the Capability Port and the Behaviour Port) through which the discovery framework can be invoked in order to deal with functional and behavioural queries.

In the scenario in Figure 6.4, providers publish WSDL, OWL-S and (semantically enhanced) WS-BPEL service descriptions by using the (existing) Publish UDDI Port. In particular, the discovery framework should access the published OWL-S and WS-BPEL descriptions in order to generate their hypergraph and OCPR net

---

\(^1\)Note, yet, that (the functional analysis of) the discovery framework of this thesis is also capable of returning minimal sets of services, while the state-of-the-art approaches (e.g., [5]) may return non minimal sets of services (as previously discussed in Subsection 3.3).
representations and store them into its own local registry. Then, the (existing) Inquiry UDDI port can be used by clients to perform keyword-based searches in the style of white/yellow/green pages. In such a case, the client requests are directly processed by the UDDI registry which accesses all the published (WSDL/OWL-S/WS-BPEL) services. Clients can also submit advanced functional and behavioural queries through the special ports Capability Port and Behaviour Port, respectively. In such a case, the functional/behavioural client requests are processed by the discovery framework that accesses only service descriptions providing both semantics and behaviour information, which are suitably represented in the local registry of the discovery framework.

A first advantage of this solution is that we can use existing UDDI registries and technologies for the submission of services, thus not asking providers for registering their services into the local (non standard) registry of the discovery framework. Another benefit of a discovery framework embedded into UDDI is that UDDI can directly publish WSDL descriptions and satisfy keyword-based client requests, so that the discovery framework can concentrate on the portion of published services which provide semantics and behaviour information in order to satisfy complex (functional/behavioural) client requests.
Bibliography


Appendix A

Proof of Theorem 4.1

In this Chapter a formal proof of Theorem 4.1 is given. Since the proof is quite long, we give a modular presentation of it, proving several intermediate steps. We first define some notational convention and then we enounce several lemmas that will be pivotal in the rest of the proof.

We say that a context $C[-]$ is compatible with an open net $\mathcal{N}$ if $C[\mathcal{N}]$ is well-defined, i.e., if the outer interface of $\mathcal{N}$ coincides with the inner interface of $C[-]$. Writing $C[\mathcal{N}]$ we implicitly mean that $C[-]$ is compatible with $\mathcal{N}$. Similarly, $\text{Op}(\mathcal{N})_f$ denotes the set $\text{Op}(\mathcal{N}) \setminus \{f\}$, implicitly assuming that $f$ is the final place of the outer interface of the OCPR net $\mathcal{N}$.

The following lemma is tacitly used during the whole Chapter.

**Lemma A.1 (contexts preserve observations).** Let $(\mathcal{N}, M)$ and $(\mathcal{N}', M')$ be OCPR nets with markings, $C[-]$ a context compatible with the two nets, and $U \subseteq \text{Op}(\mathcal{N})_f$ ($=\text{Op}(\mathcal{N}')_f$) a marking. If $\text{Obs}(\mathcal{N}, M) = \text{Obs}(\mathcal{N}', M')$, then $\text{Obs}(C[\mathcal{N}], M \cup U) = \text{Obs}(C[\mathcal{N'}], M' \cup U)$.

Hence, when proving that a bisimulation is a congruence, we just consider the dynamic part, since the lemma above ensures that the static part (represented by the condition $\text{Obs}(\mathcal{N}, M) = \text{Obs}(\mathcal{N}', M')$) is preserved under context closure.

The next lemma just states that all the internal reductions of a component net are also enabled in the composite net or, in other words, that contextual composition can not inhibit internal reductions.

**Lemma A.2 (contexts preserve reductions).** Let $(\mathcal{N}, M)$ be an OCPR net with marking, and $C[-]$ a compatible context. If $M \xrightarrow{\mathcal{N}} M_1$, then $M \xrightarrow{C[\mathcal{N}]} M_1$.

Weak bisimilarity is a congruence.

In order to prove that $\approx_W = \approx_S$, we start here by proving that $\approx_W$ is a congruence, with respect to contexts and to possible markings over the contexts.
The first step is to prove that that \( \approx_{W} \) is a congruence with respect to the addition of tokens to the open places of a net. For the sake of readability, in the following we denote \( M \setminus \{ f \} \) as \( M - f \), implicitly assuming that \( f \) is the final place of the outer interface of the OCPR net at hand.\(^1\)

**Lemma A.3** \((\approx_{W} \text{ is a congruence with respect to markings})\). Let \((\mathcal{N}, M)\) and \((\mathcal{N}', M')\) be OCPR nets with markings, such that \( O_{\mathcal{N}} = O_{\mathcal{N}'} \), and \( U \subseteq Op(\mathcal{N}) \) a marking. If \((\mathcal{N}, M) \approx_{W} (\mathcal{N}', M')\), then \((\mathcal{N}, M \cup U) \approx_{W} (\mathcal{N}', M' \cup U)\) and \((\mathcal{N}, M - f) \approx_{W} (\mathcal{N}', M' - f)\).

**Proof.** Suppose \((\mathcal{N}, M) \approx_{W} (\mathcal{N}', M')\) and let \( o \in U \). The case \( o \in M \) is obvious. If \( o \not\in M \), then \( M \xrightarrow{o}_{\mathcal{N}} M + o \); and since \((\mathcal{N}, M) \approx_{W} (\mathcal{N}', M')\), then \( M' \xrightarrow{+}_{\mathcal{N}'} M' + o \) and \((\mathcal{N}, M + o) \approx_{W} (\mathcal{N}', M' + o)\). By iterating the process, it holds \((\mathcal{N}, M + U) \approx_{W} (\mathcal{N}', M' + U)\).

Analogous reasoning holds for \( M - f \).

We now tackle the closure with respect to a compatible context.

**Lemma A.4** \((\approx_{W} \text{ is a congruence with respect to contexts})\). Let \((\mathcal{N}, M)\) and \((\mathcal{N}', M')\) be OCPR nets with markings, and \( C[-] \) a context compatible with the two nets. If \((\mathcal{N}, M) \approx_{W} (\mathcal{N}', M')\), then \((C[\mathcal{N}], M) \approx_{W} (C[\mathcal{N}'], M')\).

**Proof.** Let \( \mathcal{R}_1 = \{(C[\mathcal{N}], M + U), (C[\mathcal{N}'], M' + U) \mid (\mathcal{N}, M) \approx_{W} (\mathcal{N}', M')\} \) and \( \mathcal{R}_2 = \{(C[\mathcal{N}], M + U - f'), (C[\mathcal{N}'], M' + U - f') \mid (\mathcal{N}, M) \approx_{W} (\mathcal{N}', M')\} \) be two relations, for \( f' \) final place in \( \mathcal{N} \) and for any \( U \subseteq Op(C[\mathcal{N}]) \) marking (hence possibly including places in \( Op(\mathcal{N}) \)). We have to prove that \( \mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \) is a weak bisimulation.

Let us first consider the pairs in \( \mathcal{R}_1 \).

Let \( M + U \xrightarrow{o}_{C[\mathcal{N}]} M + U + \{o\} \). The case \( o \not\in Op(\mathcal{N}) \) is obvious. If \( o \in Op(\mathcal{N}) \), clearly \( o \not\in M \), hence \( o \not\in M' \), and also \( M' + U \xrightarrow{+}_{C[\mathcal{N}']} M' + U + \{o\} \). Since \((\mathcal{N}, M) \approx_{W} (\mathcal{N}', M')\), then \((\mathcal{N}, M + U + \{o\}) \mathcal{R} (\mathcal{N}', M' + U + \{o\}) \).

Let \( M + U \xrightarrow{-f}_{C[\mathcal{N}]} M + U - f \). The case \( f \not\in Op(\mathcal{N}) \) is obvious. If \( f \in Op(\mathcal{N}) \), then \( f \) is the final place for \( \mathcal{N} \) and \( \mathcal{N}' \), too. Since \((\mathcal{N}, M) \approx_{W} (\mathcal{N}', M')\), then \((\mathcal{N}, M + U - f) \mathcal{R} (\mathcal{N}', M' + U - f) \).

If \( M + U \xrightarrow{+}_{C[\mathcal{N}]} M_1 \), then there exists a transition \( t \in T_{C[\mathcal{N}]} \) such that \((C[\mathcal{N}], M + U)[t](C[\mathcal{N}], M_1)\), with \( t \) belonging either to \( \mathcal{N} \) or to \( C[-] \).

- \( t \in \mathcal{N} \). Then \( M + U \xrightarrow{-}_{\mathcal{N}} M_1 + U \). By Lemma A.3, \((\mathcal{N}, M + U) \approx_{W} (\mathcal{N}', M' + U)\) and then \( M' + U \xrightarrow{-}_{\mathcal{N}'} M'_1 + U \) with \((\mathcal{N}, M_1 + U) \approx_{W} (\mathcal{N}', M'_1 + U)\). By Lemma A.2, \( M' + U \xrightarrow{+}_{C[\mathcal{N}]} M'_1 + U \) and we have that \((C[\mathcal{N}], M_1 + U) \mathcal{R} (C[\mathcal{N}'], M'_1 + U)\).

\[^{1}\text{Additionally, in the proofs } M \cup U \text{ and } M \cup \{o\} \text{ denote } M + U \text{ and } M + o, \text{ respectively.}\]
• $t \in C[-]$. Then, the only tokens of $\mathcal{N}$ used to perform $t$ are those contained in the open places of $\mathcal{N}$. Since $(\mathcal{N}, M + U) \approx_W (\mathcal{N}', M' + U)$, then $\text{Obs}(\mathcal{N}, M + U) = \text{Obs}(\mathcal{N}', M' + U)$, and then the transition $t$ can be performed also by $(C[\mathcal{N}'], M' + U)$. Moreover, the firing of $t$ can produce new tokens in $\text{Op}(\mathcal{N})_f$ and possibly consume the token in $f'$. All the other either consumed or produced tokens occur in the places of $C[-]$. Thus, the states reached by $(C[\mathcal{N}'], M' + U)$ and by $(C[\mathcal{N}], M + U)$ are again in the relation $\mathcal{R}$.

For $\mathcal{R}_2$ we can proceed as above. □

The lemmas above ensure that the proposition below holds.

**Proposition A.1** $\approx_W$ is a congruence.

In other terms, $\approx_W$ is a congruence with respect to context closure and marking addition, that is, $(\mathcal{N}, M) \approx_W (\mathcal{N}', M')$ implies that $(C[\mathcal{N}], M \cup U - f) \approx_W (C[\mathcal{N}'], M' \cup U - f)$ for $f \in M$ final place of $\mathcal{N}$, and for any context $C[-]$ compatible with the two nets and marking $U \subseteq D_{C[-]} \cup C_{C[-]}$, i.e., possibly including closed places of $C[-]$.

**Weak bisimilarity is saturated.**

We open the section by defining a notion of bisimulation that is intermediate with respect to $\approx_S$ and $\approx_W$. More precisely, it differs from saturated bisimulation because it is one-step, that is, only a single firing step is considered in the bisimulation game.

**Definition A.1 (intermediate bisimulation).** A symmetric relation $\mathcal{R} \subseteq \mathcal{MN} \times \mathcal{MN}$ is an intermediate bisimulation if whenever $(\mathcal{N}, M) \mathcal{R} (\mathcal{N}', M')$ then

- $O_N = O_{N'}$ and $\text{Obs}(\mathcal{N}, M) = \text{Obs}(\mathcal{N}', M')$, and
- $\forall C[-] \cdot M[\cdot]_{C[\mathcal{N}]}M_1$ implies $M' \rightarrow_{C[\mathcal{N}]} M'_1 \not\in (C[\mathcal{N}], M_1) \mathcal{R} (C[\mathcal{N}'], M'_1)$.

The union of all intermediate bisimulations is called intermediate bisimilarity and denoted by $\approx_I$.

The proposition below is a standard result in the literature on bisimilarity.

**Proposition A.2** $\approx_I = \approx_S$.

In order to prove Theorem 4.1, we need to introduce two special contexts.

**Definition A.2** Let $\mathcal{N} = (N, O)$ be an OCPN net. Moreover, let $o \in \text{Op}(\mathcal{N})_f$ be an open place and $o'$ a data place such that $o' \not\in \text{Op}(\mathcal{N})$. Then, the contexts $\text{ADD}_o[-]$ and $\text{SUB}_o[-]$ are represented in Figure A.1.
Figure A.1: The context $ADD_A^O$ and $SUB^O$, for interface $O = (i, f, \{A, B\})$.

The structure of the net is actually irrelevant, hence the superscript recording only the interface. Intuitively, $ADD_o^O[-]$ is a context taking a net with outer interface $O$ and inserting a transition that adds a token to the place $o \in O$ and to another $o' \notin O$. The transition can be performed only once because of the control place “block” that inhibits further firings of the transition. $SUB_f^O$ has an analogous behaviour, taking a net with outer interface $O$ and inserting a transition that removes a token from the final place $f$ and adds a token to the open data place $o'$. As for $ADD_o^O[-]$, the transition can be performed only once. The data place $o'$ is used to look if the added transition have been performed. In fact, given an OCPR net $N$ with outer interface $O$ and a marking $M$, then $(ADD_o^O[N], M)$ can reach a state where $o'$ is observable if and only if it performs the new transition in $ADD_o^O$, because $o' \notin O$.

The behaviour of a net does not change, after inserting it into $ADD_o^O[-]$ and performing the added transition. This is stated by the following two lemmas.

**Lemma A.5** Let $N = (N, O)$ be an OCPR net, and let $M$ be a marking of $N$. Then, $M \rightarrow_{ADD_o^O[N]} M_1 \cup \{o, o', \text{block}\}$ iff $M \cup \{o\} \rightarrow_N M_1 \cup \{o\}$.

**Lemma A.6** Let $N = (N, O)$ be an OCPR net, and let $M$ be a marking of $N$. If $(ADD_o^O[N], M \cup \{o, o', \text{block}\}) \approx_S (ADD_o^O[N'], M' \cup \{o, o', \text{block}\})$, then $(N, M \cup \{o\}) \approx_S (N', M' \cup \{o\})$.

Analogous lemmas hold for $SUB_f^O[-]$. These results are used to prove that $\approx_S$ is a congruence also with respect to markings.

**Lemma A.7** ($\approx_S$ is a congruence with respect to markings). Let $(N, M)$, $(N', M')$ be OCPR nets with markings, such that $O_N = O_{N'}$, and $U \subseteq \text{Op}(N)_f$ a marking. If $(N, M) \approx_S (N', M')$, then $(N, M \cup U) \approx_S (N', M' \cup U)$ and $(N, M - f) \approx_S (N', M' - f)$.

**Proof.** Let $o \in U$ an open place, and let us assume that $o \notin M$. Consider the relation $\mathcal{R} = \{(N, M + o), (N', M' + o)) \mid (N, M) \approx_S (N', M')\}$: We prove that $\mathcal{R}$ is a saturated bisimulation, and then we iterate for all places in $U$. 
Suppose that \( M \rightarrow_{\text{ADD}_o[C[N]]} M_1 + o \), then, by Lemma A.5, we obtain that \( M \rightarrow_{\text{ADD}_o[C[N]]} M_1 + o + o' + \text{block} \). Since \((N', M) \approx_S (N', M')\), then \( M' \rightarrow_{\text{ADD}_o[C[N']]} M'_1 + o + o' + \text{block} \) and \((\text{ADD}_o[C[N]], M_1 + o + o' + \text{block}) \approx_S (\text{ADD}_o[C[N']], M'_1 + o + o' + \text{block})\). By Lemma A.5, \( M' \rightarrow_{C[N']} M'_1 + o \), and by Lemma A.6 we have \((C[N], M_1) \approx_S (C[N'], M'_1)\).

Analogously for \(-f\). □

**Theorem A.1** \( \approx_S \approx_W \).

**Proof.** In order to prove \( \approx_W \subseteq \approx_S \), we have to prove that \( \approx_W \) is an intermediate bisimulation. Suppose that \((N, M) \approx_W (N', M')\) and \( M \rightarrow_{C[N]} M_1 \). By Lemma A.4 we have \((C[N], M) \approx_W (C[N'], M')\), hence \( M' \rightarrow_{C[N']} M'_1 \) with \((C[N], M_1) \approx_S (C[N'], M'_1)\).

In order to prove \( \approx_S \subseteq \approx_W \), we have to prove that \( \approx_S \) is a weak bisimulation. Suppose that \((N, M) \approx_S (N', M')\).

If \( M \rightarrow_{C[N]} M_1 + o \), then \( o \) is an open place with \( o \notin M \) and thus \( o \notin M' \). Then \( M' \rightarrow_{C[N']} M'_1 + o \) and by Lemma A.7 we have \( M_1 + o \approx_S M'_1 + o \).

If \( M \rightarrow_{C[N]} M_1 - f \), we can apply an analogous reasoning.

If \( M \rightarrow_{C[N]} M_1 \), then trivially \( M \rightarrow_{C[N]} M_1 \), hence \( M' \rightarrow_{C[N']} M'_1 \) with \((N, M_1) \approx_S (N', M'_1)\). □