PROCESS BASED MATCHMAKING OF SERVICES

by

Akın Günay

B.S., Computer Engineering, Eastern Mediterranean University, 2005

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2008

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to Dr. Pınar Yolum for her invaluable guidance, great patience, and friendly and encouraging support. This thesis would not be possible without her efforts.

I would like to thank my thesis committee members Dr. Ayşe Bener and Dr. Ali Tamer Ünal for their efforts and valuable suggestions. Besides I am grateful to Dr. Suzan Üsküdarlı for our friendly, encouraging and guiding talks.

I shared a lot with people of AILAB and MAS Research Group during my study. I would like to thank all of them for their friendship and support.

My friends Bircan Tuner and Can Berkol were always with me in my most desperate times. I am grateful to them for their invaluable friendship and support.

I would like to thank my former supervisor Dr. Adnan Acan and former colleague Yüce Tekol for their guidance when I was a newbie in academia and research.

Parts of this work is published in the proceedings of EC-Web'07 and AAMAS'08 conferences. I would like to thank various blind reviewers of these conferences for their comments and suggestions to improve this work.

Finally, I am most grateful to my parents for their great love, patience, support and endless faith in me during my whole life.

This thesis has been supported by the Scientific and Technological Research Council of Turkey (TUBİTAK) 2210 National Graduate Scholarship Program. This research also has been partially supported by Boğaziçi University Research Fund under grant BAP07A102 and by TUBİTAK under grant 105E073.

ABSTRACT

PROCESS BASED MATCHMAKING OF SERVICES

With the increasing popularity of Web services, number of available services on public and business domains grows rapidly in the recent years. This growth in the number of services makes service discovery more important and challenging. The fundamental element of service discovery is service matchmaking. Service matchmaking is the process of retrieving suitable services given by service providers to satisfy service request of service consumers. The current standard service discovery mechanism provides only primitive service matchmaking methods, which are not sufficient to fulfill the requirements of todays consumers. Recent research to develop better service matchmaking methods is based on the use of semantic models of input-output interfaces of services and their use in service matchmaking. However these methods suffer from low precision due to lack of use of internal process knowledge of services in matchmaking.

In this thesis we propose two novel service matchmaking methods to achieve better precision than the state of the art service matchmaking methods. In order to achieve this goal, we claim that extensive use of semantically augmented internal process knowledge of service is necessary. Hence, in our proposed methods we use internal process models, which we markup with semantic concepts, as the core information source for service matchmaking. Our first matchmaking method uses finite state machines for service modeling and several structural and a semantic similarity metric for matchmaking. Our second method uses temporal logic for modeling and model checking techniques for matchmaking. We propose a generic service matchmaking framework to realize our proposed approaches and conduct case studies to evaluate strong and weak points of our proposed matchmaking methods.

ÖZET

SERVİSLERİN SÜREÇ TABANLI OLARAK EŞLENMESİ

Web servislerinin artan popülaritesi ile birlikte, son yıllarda kamu ve iş alanlarındaki mevcut servislerin sayısı hızla artmaktadır. Servis sayısındaki bu artış servis bulma problemini daha önemli ve zor bir hale getirmiştir. Servis bulmanın temel elemanı servis eşlemedir. Servis eşleme, servis sağlayıcılarının sundukları servisler arasından, servis tüketicisinin isteğini karşılamaya uygun olanlarının bulunup getirilmesi sürecidir. Mevcut standart servis bulma mekanizmaları günümüz kullanıcıların ihtiyaçlarını karşılamayacak basit servis eşleme metodları sumaktadırlar. Daha iyi servis eşleme metodları geliştirmek için yakın zamanda yapılan araştırmalar servislerin girdi-çıktı arayüzlerinin anlamsal modellerine ve bu modellerin servis eşlemesinde kullanımına dayanmaktadırlar. Ancak bu metodlar içsel süreç bilgilerinin yetersiz kullanımı nedeniyle düşük hassasiyet göstermektedirler.

Bu tezde mevcut servis eşleme metodlarına kıyasla daha iyi hassasiyet elde etmek amacıyla iki yeni servis eşleme metodu önerilmiştir. Bu hedefe ulaşmak amacıyla, servislerin anlamsal olarak zenginleştirilmiş içsel süreç bilgilerinin kapsamlı olarak kullanımının gerektiği öne sürülmüştür. Buna bağlı olarak, önerilen metodlarda servis eşlemenin temel bilgi kaynağı olarak servislerin anlamsal kavramlarla işaretlenmiş içsel süreç modelleri kullanılmıştır. Önerilen ilk eşleme metodunda servis modelleme için sonlu durum makineleri, servis eşleme için çeşitli yapısal ve anlamsal ölçüm metodları kullanılmıştır. Önerilen ikinci eşleme metodunda ise servis modelleme için zaman mantığı, servis eşleme için model doğrulama teknikleri kullanılmıştır. Önerilen metodların gerçekleştirilmesi amacı ile bir servis eşleme taslağı önerilmiş ve durum incelemesi yolu ile önerilen metodlar değerlendirilmiştir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ACR	Action Cover Rate
CAC	Common Action Count
ED	Edit Distance
FOL	First Order Logic
FSM	Finite State Machine
LCSeq	Longest Common Subsequence
LCStr	Longest Common Substring
OWL	Web Ontology Language
OWL-S	OWL for Services
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SWRL	Semantic Web Rule Language
LTL	Linear Temporal Logic
UDDI	Universal Description Discovery and Integration
WSDL	Web Service Description Language
XML	Extensible Markup Language

1. INTRODUCTION

Service Oriented Architecture (SOA) [1] is a distributed computing paradigm, which allows interaction between software components regardless of their platform, implementation and location through utilizing generic services as building blocks. Although the term "service" is in use since early 1990's, the SOA paradigm started to gain popularity with the standardization of Web service technologies such as Simple Object Access Protocol (SOAP) [2], Web Service Description Language (WSDL) [3] and Universal Description Discovery and Integration (UDDI) [4] in 2000's. Today, Web services [5] are the most widely used technology in the realization of SOA concepts.

In SOA there are two main entities. These are the *service provider* and the *service consumer*. A service provider is an entity that provides services to other entities. A service consumer is an entity that requires a service in order to achieve a goal. In order to use a service the service consumer should have some meta-knowledge about the service such as its capabilities and location. However in SOA service consumer does not have any prior built-in knowledge about existing services and their providers. Hence a service consumer must *discover* appropriate services and their producers in order to use these services.

Service discovery [1] is one of the key challenges in SOA. The main concern of service discovery is to find the service that can satisfy some specified properties and functionality in an accurate and efficient manner. In the core of the service discovery process there is the matchmaking method. The matchmaking method defines the details of the process to determine whether a service can satisfy the required properties and functionality or not.

Service matchmaking methods mainly use two types of information. These are the functional capabilities and non-functional properties of services [6, 7]. Functional capabilities mainly define the behavior of services. This behavior is the internal process of the service, which defines the actions performed by the service on the inputs to

produce the outputs and other effects in particular situations. On the other hand non-functional properties define the quality of the functional capabilities of services, through various ways such as the quality of service parameters, availability and performance metrics. In general, use of non-functional properties for matchmaking is simpler and clearer compared to the use of functional capabilities, since non-functional properties are easy to measure and represent. On the other hand representation of functional capabilities is a challenging issue by itself, which hardens their use in the matchmaking process. However, without using the information about the functional capabilities it is not possible to realize accurate service matchmaking, since functional capabilities describes what the service actually does.

One main challenge of the service discovery is to preserve precision and recall properties of the discovery results. In this manner the service discovery process should minimize false positives and false negatives. That is, the service discovery mechanism should find only the services that are capable of providing the functionalities and other properties requested by the consumer and eliminate the rest of the services that do not satisfy these requirements. On the other hand, the service discovery mechanism should minimize the number of services that is not involved in the discovery results, although they can satisfy the requirements of the service consumer. The challenge to preserve precision and recall arises because of the functional complexity of services. It is challenging to decide which service satisfies the requested capabilities, since services may provide various functional capabilities in different detail levels. Another challenge of the service discovery arises from the open and dynamic nature of SOA. In SOA, services may appear and disappear at any time in a system. Additionally the number of services at any time in a system is also unlimited. Therefore, the SOA environment is highly dynamic, which requires an active service discovery approach that can adopt itself to the changing environment through efficient and scalable matchmaking methods. A good service discovery approach should handle these challenges.

With the increasing popularity of the SOA, standard service discovery mechanisms are developed by the research community and industry. The current standard service discovery mechanism is the UDDI, which is a platform independent registry for

services that is based on Extensible Markup Language (XML) [8]. Although UDDI is widely accepted in the industry, it is highly static and the capabilities provided by UDDI for service discovery is primitive. The main discovery mechanism of UDDI is keyword based search, which is away from satisfying the needs of its users. The open and dynamic environment of SOA requires more sophisticated automated discovery methods to handle the requirements of the users.

In the recent years the idea of Semantic Web [9] provides researchers the facilities for automated discovery. Semantic Web is an extension of the current Web, where the Web resources and their online content are expressed not only in natural language, but also in a machine processable way in order to allow software agents to reach these resources and to read, understand and use their content. To achieve this, the resources and the content on the Web is marked-up through semantic concepts, which are presented in common ontologies. Following this vision researchers propose new service discovery methods [10, 11, 12] to overcome the weaknesses of the UDDI based service discovery. These methods mainly use the input and output parameters of services, which are related with semantic concepts from ontologies and the subsumption relations between these concepts, in order to capture the functional capabilities of services. This approach provides only a limited knowledge about the functional capabilities of services, since it treats services as black-boxes and does not consider the actual internal process of the service. This limited knowledge causes problems especially when the complexity of functional capabilities of services increases, since it is not possible to capture these functional capabilities of such services accurately, simply by checking the input and output parameters of services. For instance, assume that there is a service that requires a string input and produces a floating number output. Also assume that the input and output arguments of the service are associated with car and price concepts respectively, through using an ontology. In such a case, by checking only the input and the output of this service we can intuitively guess that the service returns the purchase price of the car, however the price type output might also be the rent or assurance price of the car. As the example demonstrated, even in such a simple case it is not possible to understand the functional capabilities of the service by only checking its input and output.

The goal of this research is to develop novel service matchmaking methods that are more active and precise than other state of the art matchmaking methods. In order to achieve this goal, we claim that extensive use of knowledge about the functional capabilities of services is necessary. Hence, in our research we propose two service matchmaking methods that are based on the use of internal process models of services. These models provide us the knowledge about the functional capabilities of services. Additionally, in order to achieve our goal, we argue that it is necessary to augment the syntactic models of services with semantic information in order to capture the full meaning of these models and to be able to perform reasoning activities on these models. In order to do this we markup service models with semantic concepts represented in ontologies. In both of our methods we use finite state machine (FSM) based techniques to model the internal processes of services, where we markup elements of these model with semantic concepts. We also develop a semantic metric, which is specialized to our design of ontologies, to capture the relations between the concepts. Addition to the matchmaking methods, in this study we propose a modular matchmaking framework. This framework allows us to use of multiple matchmaking methods in conjunction, where matchmaking methods can be manipulated in ad-hoc manner. We evaluate our methods with case studies, where we investigate strengths and weaknesses of our methods.

In our first matchmaking method we model both services and service requests as separate FSMs. In this method we mainly use various structural similarity metrics for service matchmaking. Through using these metrics our aim is to capture functional similarities between services and service request, and to determine which service or services can satisfy the functional requirements of the service consumer. To enhance our method with semantic information, we also associate the elements of the FSM models with semantic concepts from an ontology and modify the structural metrics in order to use this semantic information. Our second matchmaking method is based on the use of model checking techniques for matchmaking. In this method to model the functional capabilities of services we use a process specification language that provides us the fundamental structures to represent the details of the internal process of services. To model service requests we use a set of temporal logic formula, where each

formula represents a specific functional capability requested by the user. In our method using these service models and service requests we apply model checking techniques for matchmaking. Using model checking our aim is to capture whether the services can satisfy the functional requirements requested by the service consumer in a fine grained manner. Similar to our first method we use semantic information to improve capabilities our method. In this manner we use semantic information to generate alternative functional capabilities from the original functional capabilities in the request in order to find services that can partially satisfy the requests of service consumers.

It is important to note that, the focus of this research is on the use of functional capabilities of services for service matchmaking. Therefore, we ignore the use of non-functional properties of services for this purpose. However, it is simple to integrate non-functional properties of services with our proposed matchmaking methods.

The rest of this thesis is organized as follows: Chapter 2 provides the required background knowledge related to our research including a detailed definition of the service matchmaking problem and the relevant techniques and technologies that are used in our research. Chapter 3 discusses the key issues in the description of services and service requests especially based on the use of semantic knowledge. Chapters 4 and 5 explain our proposed matchmaking methods in detail. Chapter 6 provides the details of our general matchmaking framework that can accommodate any matchmaking method. Chapter 7 presents the results of our case studies and a comparison of our proposed matchmaking methods. Chapter 8 discusses the related work in comparison to our methods and concludes this thesis.

2. BACKGROUND

2.1. Service Oriented Architecture and Web Services

SOA is a distributed computing paradigm [1]. SOA provides an architecture that involves all aspects of creation and use of business processes, which are represented as services, in a platform independent and loosely coupled manner. In SOA functionality is decomposed into composable and reusable services that can be distributed over a network.

The goal of SOA is to make it possible to create large applications with complex functional capabilities through composition and orchestration of already existing services. To achieve this goal reusability, granularity, modularity, composability and interoperability are embraced as the guiding principles by SOA.

Although SOA can be realized through various approaches, use of Web services is the most widely accepted and practiced way of this realization. A Web services [5] is a piece of software that provides some well defined functionality over a network in a machine independent manner. In a Web service based realization of SOA there are three types of entities. These are the service consumer, the service provider and the service registry. The service provider is the entity that provides one or more services to service consumers. The service consumer is the entity that requires a service for its own purposes. The service registry acts as a middle-agent [13] in order to guide service consumers and service providers to find each other. In this manner it accepts service advertisements from service providers and service requests from service consumers and using these two information sources it guides the other entities. Although there are different efforts to develop standards for Web services, recently there are three standards, which are widely accepted and implemented by the Web services community. These are the SOAP, WSDL and UDDI. SOAP is and XML based protocol to exchange messages over a network in a machine independent manner. WSDL is a description language to describe the properties of services in a machine processable manner. UDDI

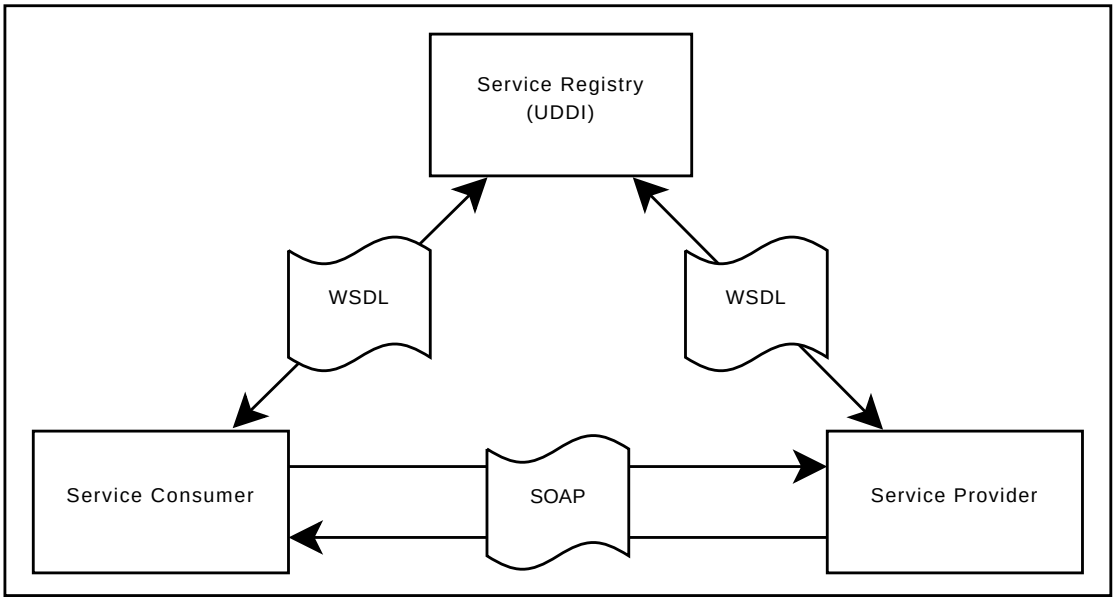

Figure 2.1. The Web service life cycle

is a registry where service providers can advertise their services to find potential service consumers.

Figure 2.1 shows the Web service life cycle and the use of the standards in this life cycle. This life cycle works as follows: First the service provider sends the description of its service as a WSDL file to the UDDI service registry. The UDDI service registry stores this description with the description of other existing services in its service database. When a service request is made by a service consumer, which is in WSDL, the UDDI service registry searches its database and try to find a service that can satisfy the request of the user. If the UDDI service registry finds one such service, it sends the UDDI description of this service back to the service consumer. Then using this WSDL service description the service consumer make the service call as a SOAP message. After receiving the service call the service provider executes the service and returns the result of the service to the service consumer again as a SOAP message.

2.2. Service Matchmaking Problem

Service matchmaking is the process of finding one or more services from a set of existing services that matches to a given service request. There are three entities

in service matchmaking. These are the service provider, the service consumer and the service matchmaker. The service provider provides one or more services to service consumers. Usually there are several service providers in the environment. In order to reach potential service consumers, service providers advertise their services publicly. The service consumer is the entity that looks for some services to satisfy its needs. To find services, service consumers make service requests to the service matchmaker in which service consumers describe the functional capabilities and non-functional properties they require from the service. Functional capabilities defines the requested behavior of the service. Non-functional properties defines the restrictions of the user over the functional capabilities provided by the service. The matchmaker acts in between the service consumer and the service provider as a middle-agent [13]. It collects the service advertisements published by the service providers and matches them to the request made by service consumers.

Formally, we can define the matchmaking problem as follows; in service matchmaking there is a set of service providers $sp_1, sp_2, ..., sp_n$, a set of service consumers $sc_1, sc_2, ..., sc_3$ and the matchmaker sm. Each sp_i provides a set of services denoted by sp_{i_s}. The set S is the global set of services and includes all the services provided by all the providers. The service consumers generates service requests denoted by sc_{i_r} to represent their service request. The set R is the global set of requests and includes all the service requests made by all the consumers. The task of the service matchmaker is to find one or more services s from S for each request r in R, where the particular selected service s satisfies the requirements of the associated r.

The matchmaker uses a matchmaking algorithm (method) to match service consumer requests to the services given by the service providers. Hence the matchmaking algorithm is one of the most important component of service discovery. This matchmaking algorithm may use various type of information sources like functional capabilities and non-functional properties of the service.

There are three properties expected from a good matchmaking algorithm [10].

- The matchmaking algorithm should minimize false positives and false negatives. By false positives we mean the services that are not capable of providing the required functionality, however determined as a match for the service request by the matchmaking algorithm. By false negatives we refer to the services that are capable of providing the required functionality, however not determined as a match for the service request by the matchmaking algorithm.
- The matchmaking algorithm should support flexible matches. That is it should return partially matching results addition to the exact matches. These partial matches might not satisfy all the requirements of the service consumer, but they might be useful in various conditions. However, there should be a control mechanism to manage the degree of flexibility of the matchmaking algorithm.
- The matchmaking process should be scalable and efficient, and completed in a reasonable amount of time. This property is important because of the open nature of the service environment, where the number of services is unlimited.

2.3. Current Service Matchmaking Architecture

The current service matchmaking architecture for Web services is based on WSDL and UDDI standards. WSDL is an XML based description language for services. In WSDL services are defined as a collection of network endpoints, which called as ports. WSDL defines services through abstract interfaces that can be bind to concrete instances and protocols, which allows reuse of these abstract interfaces. UDDI is an XML based registry, where service providers can advertise their services. A UDDI registry consists of three main components. These components are called as White, Yellow and Green Pages. White pages provide basic information about service providers, such as address and contact information. Yellow pages provide industrial categorizations of services based on standard taxonomies. Green pages provide the technical details of services.

The service discovery mechanism of UDDI is not sophisticated and requires extensive human interaction. For service discovery UDDI provides only a keyword based search facility and left the rest of the service discovery process to the service consumer.

In this sense the main weakness of the UDDI and WSDL based service discovery architecture is the lack of facilities for automation of the service discovery process. Although WSDL is machine processable, since it is based on XML, it does not contain any semantic information, which requires human interpretation of the provided information to make the meaning of this information clear. On the other hand UDDI provides only a keyword search, which is far away to support the requirements of the automated discovery.

2.4. Techniques

2.4.1. Finite State Machine

An FSM [14] is a formalism to capture the flow of processes. Using FSMs we can model the fundamental control structures (sequences, choices and loops) of process flows. Definition 1 gives the formal definition of an FSM.

Definition 1 *A finite state machine is a 5-tuple (Q, Σ, δ, q_0, F), where*

- Q *is a finite set called states,*
- Σ *is a finite set called alphabet,*
- $\delta : Q \times \Sigma \rightarrow Q$ *is the transition function,*
- $q_0 \in Q$ *is the start state, and*
- $F \subseteq Q$ *is the set of final states.*

A is the language (the set of all strings) that machine M recognizes and we show this by A $L(M)$.

States in FSM represents the current status of the system considered by the FSM. Each FSM has an initial state, which is the entry point of the underlying system and a set of final (accept) states that represent the regular endpoints of the underlying system. The transition function defines the rules of moving from one state to another

according to the input (action) represented as a letter in the alphabet.

2.4.2. Linear Temporal Logic

Linear Temporal Logic (LTL) [15, 16] is a temporal logic, where the future is seen as a sequence of states or simply as a path. LTL formulae are built up from a set of proposition variables, the usual logic connectives and four temporal modal operators X, F, G and U. X stands for *next*. It is a unary operator and it means that its bounded proposition must hold at the next state of the given path. F stands for *eventually*. It is another unary operator and it means that its bounded proposition must hold eventually at some future state(s) of the given path. G stands for *globally* and it is a unary operator. It means that its bounded proposition must hold at all future states of the given path. U stands for *until* and it is the only binary operator. It means that the first proposition bounded to U must hold until the second proposition starts to hold. U also requires that the second proposition must hold in some future state.

In LTL the underlying structure of time is modeled as a transition system. A transition system consists of a set of states and transitions. Each state is associated with atomic propositions that are true in this particular state. The transitions determine how the system evolves from state to state. We call a transition system as model in the rest of this section.

Definition 2 *A transition system* $M = (S, \rightarrow, L)$ *is a set of states* S *and a transition relation* $\rightarrow$*, such that every* $s \epsilon S$ *has some* $s' \epsilon S$ *with* $s \rightarrow s'$*, and a labeling function* $L : S \rightarrow P(Atoms)$.

In a transition system it is possible to initiate more than one transition from a state, therefore the transition system may have more than one path. This allows us to represent various possible futures.

Definition 3 *A path in a model* $M = (S, \rightarrow, L)$ *is an infinite sequence of states*

$s_1, s_2, s_3, ...$ in S such that, for each $i \geq 1$, $s_i \rightarrow s_{i+1}$. Than we can write the path as $s_1 \rightarrow s_2 \rightarrow ....$

An LTL formula is formed from a set of propositional variables $p, q, ...$, logic connectives $\neg, \vee, \wedge, \rightarrow$ and temporal connectives X, F, G, U.

Definition 4 *$M = (S, \rightarrow, L)$ is a model, $s \epsilon S$ and ϕ is an LTL formula. We say $M, s \models \phi$ if, for every execution path π of M starting at s, we have $\pi \models \phi$.*

2.4.3. Model Checking

Model checking [15] is a method to formally verify distributed systems and protocols. In model checking the aim is to verify that certain properties are hold for a system. To verify a system, the model checker exhaustively checks all possible executions of a system against the specified properties. In general the properties are described using temporal logic.

Model checking process has three main steps. The first step is to decide which properties of the system are going to be verified. Generally, two types of properties are verified in model checking. These are safety properties, which ensure that the system will never reach to an undesirable state like a deadlock, and liveliness properties, which ensure that the system reach to the desired states in its execution.

After the decision about the properties to be verified is made, in the second step a model of the system is created. The important issue in this step is to decide which behavior of the system should be modeled and what type of abstractions will be made accordingly to hide irrelevant behavior and details of the system, in order to successfully check the properties defined in the first step. This is important to reduce the complexity and make it possible for the model checker to exhaustively analyze all possible executions of the system, which might be impossible when the system is fully modeled.

The last step is the use of a concrete model checker to verify the properties of the system. The result is true if the model entails the properties and false otherwise. In the case of failure, the model checker also provides a trace of the execution, which leads the system to an undesired state.

2.5. Technologies

2.5.1. Ontologies and Web Ontology Language

An ontology is a data model that represents a domain [17]. An ontology consists of a set of concepts and the relations between these concepts. An ontology is generally used to reason about objects in the domain modeled by the ontology. In an ontology there are mainly four type of elements. These are *individuals*, *classes* (also called concepts), *attributes*, and *relations*. Individuals are the base level components of an ontology. Individuals can be concrete classes such as people, animals and cars or they can be abstract such as numbers or words. Although one of the main purpose of ontologies is to classify individuals, it is not necessary for an ontology to represent individuals explicitly as part of the ontology.

A class defines an abstract set of objects that share common properties. For instance class Vehicle defines all the vehicles and class Car defines all the cars. In ontologies classes can subsume other classes. For instance Vehicle class subsumes Car class. This means that Car class is a specialization of Vehicle class, where the latter class is a member of the former and has all of its properties. However, addition to the properties of the former class the latter class has some additional more specific properties. Using these subsumption relations hierarchies of classes can be created, where the classes turn from general to specific when it goes from top to bottom on the ontology.

In the ontology the properties of the individuals are represented by the attributes associated with the class that the individual belongs. Each attribute has a name and an assigned value. For instance the car class may have a number of doors attribute and

the individual of the car class "Nissan Micra" has value 4 for this attribute. Addition to simple data types like integers, attributes can have complex data types as their values. Ontologies without attributes are called taxonomies.

Much of the expressive power of ontologies comes from the relations between the classes, where these relations define the semantics of the modeled domain. A relationship is actually an attribute, which has an another class as value instead of a data type. One of the most fundamental relationship in an ontology is the subsumption relation. For instance, the subsumption relation can be represented as Vehicle is-superclass-of Car or the converse Car is-a Vehicle. Although subsumption is the most common relationship type in an ontology, most of the time ontologies involve many other type of relations to model the domain deeply.

The Web Ontology Language (OWL) [18] is a language to define Web ontologies. An OWL ontology consists of classes, properties and instances of the classes. These constructs define the semantics of the domain that is modeled by the ontology. OWL allows user to define classes by declaring simply a name and their individuals by given the class that it belongs. OWL also supports more complex mechanisms to define classes like set operators and enumerated classes. In OWL there are two types of properties. These are the datatype and the object properties. Datatype properties are relations between instances of classes and datatypes included by XML schema. Object properties are relations between instances of two classes. In OWL properties may have characteristics like transitivity and symmetry and restrictions like cardinality.

OWL has three sub-languages OWL Lite, OWL DL and OWL Full. From these three languages OWL Lite is the less expressive one. However it still provides enough expressive capabilities in most of the cases like creating taxonomies. The advantage of OWL Lite is its low computational complexity. OWL DL provides the maximum expressive power while preserving completeness and decidability properties of the language. OWL Full provides maximum expressiveness sacrificing completeness and decidability.

2.5.2. OWL-S: Web Ontology Language for Services

OWL-S [19] is an ontology build on top of OWL to describe Semantic Web services. OWL-S provides facilities for automatic discovery, invocation, composition and monitoring of services by users and also by software agents.

OWL-S has three main constructs to define three essential type of knowledge about a service. These constructs are the service profile, the process model and the service grounding.

The service profile provides information about what the service does. It provides basic catalog information like the name of the service, a textual description, service provider and category of service. It also provides information about the input-output interface of the service and the required preconditions and the effects after the service is executed.

The process model of OWL-S gives a detailed description of the service through modeling it as an abstract process. This model explains how to interact with the service in a step by step manner, where each step is defined using the preconditions and inputs required to execute this step and outputs and effects produced by the execution of this step. A process is called as an atomic process if it can be performed in a single interaction. Through combining atomic processes with control structures such as sequence, split and choice it is possible to compose composite processes, which requires multiple interactions.

The service grounding provides the information about the concrete implementations described in the service profile and process model constructs, such as the network locations of services and protocols required to interact with the described services.

3. KNOWLEDGE REPRESENTATION

There are two main information sources for Web service discovery. These are the *service descriptions* created by service providers to define the capabilities of the services that they provide, and *request descriptions* created by service consumers to define their expectations from services. In this chapter we discuss properties of these descriptions and how we model the information extracted from these descriptions.

3.1. Service and Request Descriptions

Service and request descriptions are the two fundamental information sources for Web service discovery. A service description provides information about the functional capabilities [6] and non-functional properties [7] of a service. Similarly, a request description defines the requested functional capabilities and non-functional properties by the service consumer from the service. Functional capabilities of a service defines the behavior of the service in terms of what it can do. That is the internal process of the service, which defines step by step the particular actions performed by the service on the inputs to produce the outputs and other effects for all possible executions. Non-functional properties are aspects that define qualities of the functional capabilities or in other words constraints over the functional capabilities. Some typical non-functional properties are availability, quality, price and security. In this thesis our main focus is on the functional capabilities of services and functional requirements of the service consumers for Web service discovery.

There are various approaches to describe services as explained in Chapter 2. However most of these approaches provide similar types of information structures to define the functional capabilities of services. First and most frequent of these structures is the input-output interface of services. Input-output interface of a service defines the inputs required by the service from the consumer to execute the service properly and the outputs produced as the result of the service execution. Second type of structure involves the pre-conditions and effects of the service. Pre-conditions are the conditions

required by the service to start the execution. Effects show how the state of the world is changed after the execution of the service. The last information structure is the service model. The service model describes the internal process of the service as a flow considering all possible executions and their outcomes.

One important distinction between the service description approaches is the use of semantic information. Early approaches such as WSDL ignore semantics and concentrate only on the syntax [20]. These approaches provide only a set of syntactic structures like types and variable names where the meaning of these structures are defined elsewhere or like in most of the cases their meanings are left to the intuition of the consumer. This approach has two main drawbacks. First, it limits automation of the discovery process, since it is hard (mostly impossible) for machines to interpret syntactic information without any human interaction. Second, even with human interaction, since the semantics of the syntactic structures are not clear, the result quality and performance of the service discovery process is poor.

The following example explains the importance of semantic information. Assume that there is a service, which takes the name of a city and returns the current air temperature of this city in degree Celsius. By using only syntactic information, we only now that the service takes a string input and returns a decimal output. From this syntactic point of view there is no way to understand that the input is a city name and the output is the air temperature. However by annotating the input and output arguments with semantic information, their meaning is clear without any ambiguity both for human and machine processing.

3.2. Semantic Information and Ontologies

As we demonstrated in the previous section, use of semantic information improves capabilities of service discovery approaches. However, we need a methodology to formally define this semantic information and ontologies constitute a suitable formalism for this purpose. An ontology [17] is simply a conceptualization of a domain. It defines the concepts and instances of concepts in a domain and also the relations between these

concepts.

For instance the concept of a city can be captured by an ontology to define semantics of the geography domain. This ontology may further involve concepts like region, country, continent and so on and these concepts may have relations between them. For instance a city exists in a region and a region exists in a country. These relations makes it possible to perform reasoning operations on the ontology to deduce further information. For instance if we know that city of Istanbul exists in the region of Marmara and the region of Marmara exists in the country of Turkey we can easily deduce that the city of Istanbul exists also in the country of Turkey.

In addition to defining the meaning of concepts the knowledge obtained from the ontology can be used to enhance service discovery process. For instance, assume that there is no service that provides the requested capabilities in the city temperature example given in Section 3.1. However there might be a service that provides the temperature values of all cities in a region. Since we can deduce the relation between a city and a region via the ontology, we can offer to the consumer this region temperature service as an alternative to the city temperature service.

3.3. An Action as an Abstract Concept

As we explained before, service description approaches use various information structures like input-output interfaces, pre-conditions and effects, and service models to describe functional capabilities of services. To enhance service descriptions and these information structures with semantic information, they must be represented in ontologies. Such a representation requires use of various concepts for services and each information structure. Additionally several relations between these concepts must be established, which increases the complexity of the ontology. Although by using this approach a service can be described in great detail, because of the complexity of the ontology, performing reasoning operations require more computation. In this thesis, to overcome this challenge, we unify these information structures under an abstract concept that we call *action*. An action is an activity on some given input(s), where

some output is produced as the result of this activity. An action may also have its pre-conditions and effects. Abstraction level of an action depends on the domain in consideration. For instance, in a domain, an action can be a single operation that changes a bit in the memory, while in another domain, it can be a set of operations to buy a book online. An action may represent a part of a service or a complete service. The important issue is to hold the rule that each action must be performed atomically in the corresponding domain. Using this approach we are able to represent services or part of services as stand alone concepts in ontologies and therefore simplify the complexity of ontology model and reasoning operations.

3.4. Description of the Action Ontology

In order to use the action concept in matchmaking we design an action ontology, where we model actions and the relationships between them. In this action ontology we associate each action with a concept and use is-a relations to create a hierarchy structure between the actions. In this hierarchy the actions on the top levels are more general actions and if we go down by following the relations we reach more specialized actions.

Assumption 1 *A parent action can handle all the functional capabilities of all of its child actions.*

We make the Assumption 1 in order to unify the relations and to preserve the tree structure of our action ontology. To make this assumption clear, consider the following example. For instance, if action A has two child actions as A_{c1} and A_{c2}, then we assume that action A can provide the functionality of both A_{c1} and A_{c2}.

Figure 3.1 is a portion of a sample action ontology that we designed for e-commerce domain. In this domain, there are basic actions like, searching for commercial goods, adding goods to shopping cart, purchasing goods, making payment for purchased goods and so on. In the sample ontology these actions are represented by

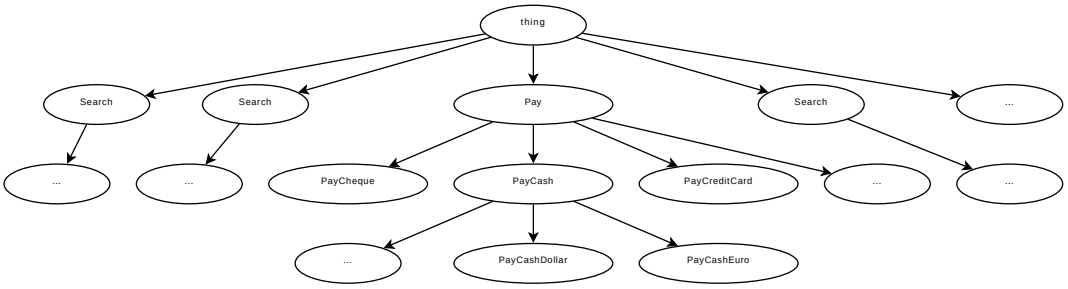

Figure 3.1. Portion of the action ontology for e-commerce domain

high level concepts `Search`, `Add`, `Purchase`, `Pay` and so on. Child concepts of these high level concepts represent more specialized versions of the high level actions. For instance `pay` action has child actions `PayCreditCard`, `PayCash` and `PayCheque`. `PayCreditCard` and `PayCash` actions have further specialized child actions. This level of specialization is not limited and it depends on the domain under consideration.

Let us now discuss the Assumption 1 on this concrete example . Here, the `Pay` action is a high level action that has several more specialized actions as child actions. According to our assumption, `Pay` action can provide all the functional capabilities provided by its child actions. That is, if we have `Pay` in a service description, then we can deduce that this service can provide us any type of payment action. Similarly, if the `Pay` action is required in a service request, we can deduce that the service consumer is convenient with any type of payment action. On the other hand, `PayCash` is a specialized payment action. That is, if we have `PayCash` in a service description, then this service can only provide this specific payment action. Therefore it can handle a request that involves `Pay` action, which is more general, only partially, since it can not provide all the required payment actions. In the case of service request, this issue is different. If the service request involves the specific payment action `PayCash` and the service involves the more general `Pay` action, then the service can handle the requested payment action, since general action `Pay` can provide all the functional capabilities of its more specialized actions.

4. FSM BASED MATCHMAKING METHOD

In this chapter we present our first service matchmaking method. In this method we use FSMs to model both services and requests. We mainly use four structural similarity metrics to matchmake services to requests. We also use an action ontology and a semantic metric to enhance our method with semantic information.

4.1. Service and Request Modeling through FSM

In this method we model services and service requests as separate FSMs. Each FSM model consists of several transitions, where each transition is associated with an action from an action ontology. We use FSM states to control the flow of actions. Figure 4.1 shows an example e-commerce service modeled as an FSM. In the figure the circles are the states. Each state is associated with a unique number for identification. The arcs between the states represent the transitions, which are actually actions from an action ontology. For simplicity we present only the name of the action and skip the rest of the details.

Using this service a consumer can search for commercial goods, add found goods to a cart, order the goods in the cart and pay the price of the ordered goods. Verbal interpretation of the model of this service is as follows. Initially in state 0, there is no connection between the consumer and the service. To establish the connection to the service, first the consumer performs the `connect` action and the service goes to the state 1. After that the consumer performs a search action to find the goods that she wants to order. This is represented by the action `searchGood`. As the result of the search action the service reaches to the state 2. In this state, the consumer can perform one of the three possible actions, according to the result of the recently performed search action. If the consumer finds the good that she looks for in the search result, she can add the this good to her cart by performing the `addGoodToCart` action. In this case the service state changes to 3. If the consumer does not find the good that she looks for in the search result, she may want to make a new search. In this case she can perform the

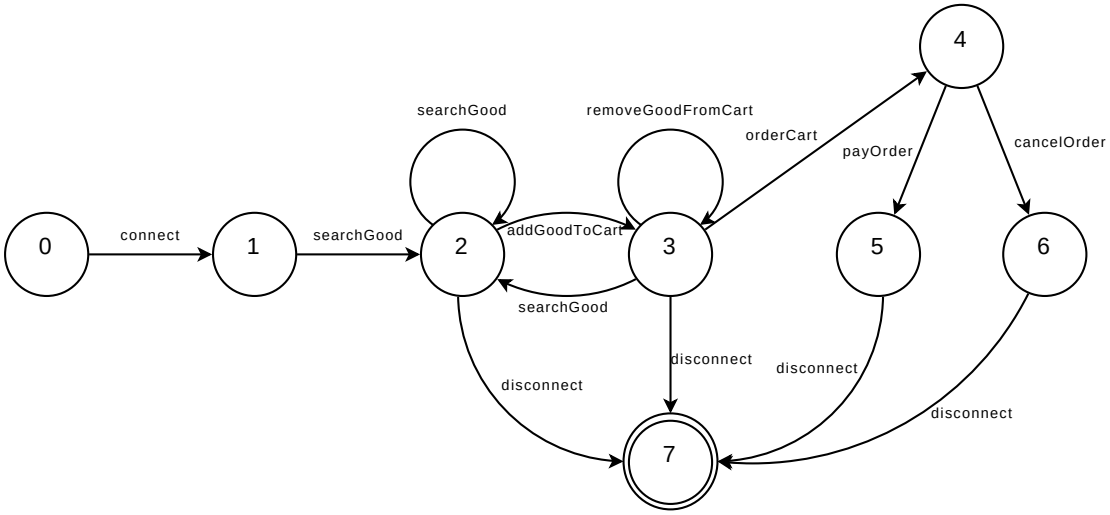

Figure 4.1. FSM model for a service in the e-commerce domain

`searchGood` action repetitively until she finds the appropriate good. This action does not change the state of the service. In the last case, if the consumer does not want to search further and decides to leave the service, she can perform the `disconnect` action and move to state 7, where the communication between the consumer and the service ends. Now assume that at this point the consumer finds a good and decides to add it to her cart and performs `addGoodToCart` action. As the result of this action, the state of the service changes from state 2 to state 3. In state 3 the consumer can do one of the four possible actions. In the first case, she can search for other goods by performing the `searchGood` action again and move to state 2. In the second case, she can remove a good from her cart by performing a `removeGoodFromCart` action. In this case the state of the service does not change. In the third case, she can order the goods in her cart by performing the `orderCart` action. In this case the state of the service changes from 3 to 4. In the last case, she can discard the goods in the cart and leave the service by performing the `disconnect` action, where the state of the service changes to 7. Assuming that the consumer decides to order her cart we continue from state 4. In this state the consumer can make the payment for her order to buy the goods by performing the `payOrder` action and move to state 5 or cancel her order by performing `cancelOrder` action and move to state 6. In either case, the consumer can only perform a `disconnect` action and the state of the service changes to state 7 where the communication between the service and the consumer ends.

As we demonstrated in the example FSM model of the e-commerce service, FSM formalism provides us the fundamental structures (looping, branching and so on) that we need to model services and requests. Additionally, because of the simplicity of the FSM models it is easy to process them, which decreases the computational cost of the matchmaking operation.

To represent FSM models in the implementation level, we use the process model component of OWL-S. The process model of OWL-S supports all the control structures, such as sequences, choices and loops, required to represent our FSM models of services. Additionally, OWL-S provide us facilities to represent semantic information, which allows us to use this information with our matchmaking method.

4.2. The Matchmaking Method

As stated in the previous section, in our method we use FSMs to model services and requests. Therefore when we matchmake a service to a request, we actually check the equality of the FSM model of the service and the FSM of the request. The formal definition of FSM equality states that two FSMs are equal to each other if they recognize the same language. In our context this definition is restrictive because of the following reasons: First, the equality definition requires that the FSM model of the service and the FSM model of the request matches exactly to each other. That is both FSMs must involve the same actions in the same order. However there might be two services that perform exactly the same task, but through different actions or through same actions but with different order. This issue cannot be captured by the formal definition of the FSM equality. Second, a service might provide more capabilities than the request. In this case the formal definition of FSM equality cannot be used. Third issue is related to flexibility of the matchmaking method. In service matchmaking the desired behavior from the matchmaking method is to match alternative or partially matching services in addition to exactly matching services for the requests. However using the formal definition of FSM equality this is not possible.

Because of these reasons we need a more flexible matchmaking method. The

```
1: reqSeqList := GENERATE-ALL-SEQUENCES(r)
2: servSimList := {∅}
3: for all s ∈ S do
4:    servSeqList := GENERATE-ALL-SEQUENCES(s)
5:    simSum ← 0.0
6:    seqCount ← 0
7:    for all reqSeq ∈ reqSeqList do
8:       seqSimList := {∅}
9:       for all servSeq ∈ servSeqList do
10:         seqSim := SIM-METRIC(reqSeq, servSeq)
11:         seqSimList := seqSimList ∪ seqSim
12:      end for
13:      simSum := simValueSum+ MAX(seqSimList)
14:      seqCount := seqCount + 1
15:   end for
16:   servSimList := serSimList ∪ (simSum ÷ seqCount)
17: end for
```

Figure 4.2. FSM based matchmaking algorithm.

matchmaking method should be able to find partially matching services addition to exactly matching services. Further, the matchmaking method should assign a numeric value between services and requests to show the degree of match. To achieve these goals, we developed a matchmaking method, where we use structural and semantic similarity metrics in combination.

We present the main algorithm of our matchmaking method in Algorithm 4.2. This algorithm takes an FSM model r for the request and a set of FSM models S, where each FSM model $s \epsilon S$ represents a service. In the algorithm, before starting the matchmaking operation, we first generate all action sequences for the request r through a procedure we call `GENERATE-ALL_SEQUENCES` (line 1). We explain the details of this procedure in Section 4.2.1. Then for each service s in the set of services S we compute an overall similarity value between service s and request r (lines 2-17). To do this,

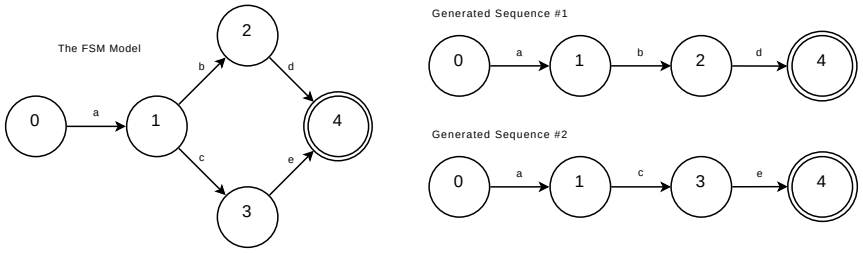

Figure 4.3. Action sequence generation from an FSM without loop

first we generate all action sequences for service s using the same procedure for the request (line 4). Then for each action sequence generated for request r we compute the similarity of this sequence to each action sequence of service s using a similarity metric (line 10) as we explain in Section 4.2.2 and we save the action sequence of s with maximum similarity value (line 13). To compute the overall similarity value of the service s to the request r, we compute the normalized sum of the maximum similarity values that we stored before. We interpret these computed overall similarity value between the service and request as the degree of match of the service for the request. At this point it is also important to note that according to the Algorithm 4.2 sequence generation process in step 4 is repeated for all services for each request. We present this step only to preserve the completeness of the algorithm. However in the implementation sequence generation for services is performed only once in an off-line manner for performance considerations.

4.2.1. Action Sequence Generation

We apply this operation to generate all action sequences that are involved by an FSM, which we use in our matchmaking method to compute the overall degree of match of service and request FSMs. To do this generation, we expand an FSM from the start state up to all final states by creating a new sequence at each branching point. Figure 4.3 shows an example for this case. This operation can be implemented through a simple recursive algorithm if the FSM structure is an acyclic directed graph without loops. But if we introduce loops to the FSM, the graph turns to a cyclic directed graph where the number of possible sequences is infinite. To handle this case we modify the expansion algorithm so that it can detect loops and stop expanding a sequence when

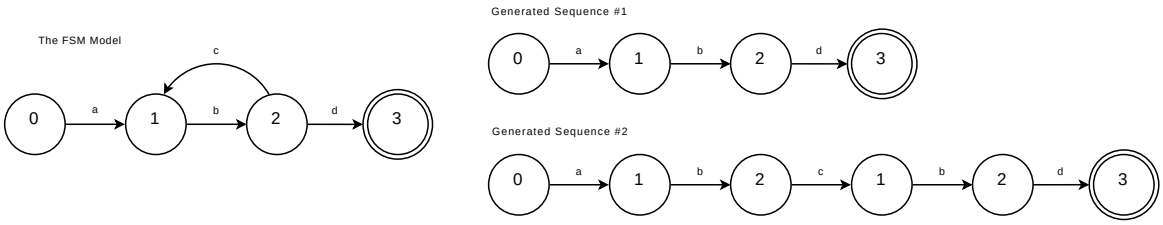

Figure 4.4. Action sequence generation from an FSM with loops

the same loop occurs more than once in it. In this way we can still capture all the information that we can obtain from an FSM, while we can deal with infinite loop situations. Figure 4.4 shows an example for this case.

4.2.2. Similarity Metric Computation

To determine the similarities and assign a numeric similarity value between sequences generated by the action sequence generation we use several structural similarity metrics in conjunction with a semantic similarity metric. We use the results of theses similarity metrics later to compute overall similarity values and hence the degree of match between requests and services. For now, it is enough to understand that, when we compute a similarity metric with two action sequences, we obtain a numeric value that shows the similarity of functional capabilities between these action sequences. We explain these metrics in Section 4.3 in detail.

4.3. Similarity Metrics

In this study we use several metrics and evaluate their advantages and disadvantages in various situations. First we use structural metrics, where the structural similarities between the sequence actions are favored. In these metrics we do not consider the meaning of the actions, that is if an action in one sequence is syntactically equal to the action in the second sequence we count them as equal, however if their names are not equal we do not consider them semantically and simply count them as not equal. This syntactic equality does not allow us to capture the full meaning of service and request models and lower the result quality of our method. Hence, we modify the structural metrics with a semantic similarity metric in order to use semantic

information provided by action ontologies and improve the result quality.

4.3.1. Structural Similarity Metrics

To compute the structural similarity between the service and request sequences in Algorithm 4.2 we use four structural metrics. These metrics are *common action count* (CAC), *longest common substring* (LCStr), *longest common subsequence* (LCSeq) and *edit distance* (ED).

Common Action Count Metric The CAC metric calculates the number of actions that are common both in the request and in the service sequences, without regarding to the order of actions and normalizes the count with the total number of actions in the sequences. The underlying intuition is that when the number of common actions of the service and request sequences increases, the two sequences are more similar to each other. The similarity between request sequence r and the service sequence s is computed as follows;

$$sim(r, s, CPC) = \frac{2N_a}{N_r + N_s} \tag{4.1}$$

, where N_a is the number of common actions and N_r is the number of actions in request r and N_s is the number of actions in service s.

Longest Common Substring Metric The LCStr metric [21] finds the longest common contiguous substring of two strings. In our context, this corresponds to the number of contiguous actions between the request and service sequences. Formally, using LCStr metric, similarity between request sequence r and service sequence s is computed and normalized as follows;

$$sim(r, s, LCStr) = \frac{length(LCStr(r, s))}{length(r)} \tag{4.2}$$

where $length(r)$ is a function that returns the length of the sequence r. $LCStr(r, s)$ is

a function that returns the common longest substring of r and s.

Longest Common Subsequence Metric The LCSeq metric [21] finds the longest common subsequence (may not be contiguous) of two strings. In our context, this corresponds to finding the number of common actions between a request and a service sequence without considering the contigiousness. Formally, using LCSeq metric, similarity of request sequence r and service sequence s is computed and normalized as follows:

$$sim(r, s, LCSeq) = \frac{length(LCSeq(r, s))}{length(r)} \tag{4.3}$$

where $length(r)$ is a function that returns the length of the sequence r. $LCSeq(r, s)$ is a function that returns the common longest subsequence of r and s.

Edit Distance (Levenshtein) Metric Edit distance [21] metric computes the minimum number of operations needed to transform one string into another, where an operation is an insertion, deletion, or substitution of a single character. In our context the strings are the sequences and the characters are the actions. We formally calculate the similarity of the request sequence r and service sequence s as follows:

$$sim(r, s, ED) = 1 - \frac{ED(r, s)}{Max(length(r), length(s))} \tag{4.4}$$

where $ED(r, s)$ is the function that computes the edit distance of the sequences and $Max(l_1, l_2)$ is the function that returns the maximum of two integers.

4.4. A Semantic Similarity Metric for the Action Ontology

One of the useful information that we can deduce from our action ontology is the semantic similarity relations between the actions. These similarity relations play an important role in the service discovery process, since they allow us to find alternative services. For instance in our action ontology there are various payment actions like

`pay_by_cash`, `pay_by_creditcard` and so on. Although these payment actions have differences, fundamentally all of them are payment actions and therefore they are similar to each other and might be interchangeable in some situations. When there is no available service with the requested payment action, using this similarity information, we can offer alternative services with different payment actions to the consumer.

There are two possible approaches to establish the semantic similarity relations between the actions in the ontology. The first approach is creating these relations by hand at the design time of the ontology. To realize this approach, the designer of the ontology must define all the similar concepts for each concept one by one, which is an exponentially growing process. Therefore this approach is not scalable. The second approach is creating these relations automatically by using a semantic similarity metric that measures the semantic similarity between two concepts in the ontology. In addition to determining the similarity between two concepts, semantic similarity metrics mostly associate a semantic similarity value between the concepts, which shows the strength of the similarity relation between the concepts.

In this thesis we propose a semantic similarity metric that is especially designed to work with our action ontology, by emphasizing the fact that a more general action can handle the tasks that can be performed by its more specialized actions. We call our metric as *action cover rate* (ACR). The intuition behind this metric is not based on the symmetric similarity relation of two actions. Instead, through this metric we try to capture an actions ability to handle another actions functional capabilities. Using the ACR semantic similarity of two concept c_1 and c_2 is computed as follows:

$$ACR(c_1, c_2) = \begin{cases} 1, & \text{if } c_1 \supseteq c_2 \\ \theta^{\|c_1, c_2\|}, & \text{if } c_1 \subset c_2 \\ \gamma^{\|R, c_1\|}, & \text{if } c_1 \not\supseteq c_2 \text{ and } c_1 \not\subseteq c_2 \text{ and } R \supset c_1 \text{ and } R \supset c_2 \\ 0, & \text{otherwise} \end{cases} \tag{4.5}$$

where θ and γ are two control parameters in the range [0,1] and $\|c_i, c_j\|$ is the arc distance in the ontology between the concepts c_i and c_j. Once more, it is important to note that ACR metric is not symmetric, that is $ACR(c_i, c_j) \neq ACR(c_j, c_i)$.

We will explain the details of each case of the ACR metric with examples from our e-commerce action ontology. In the first case, the first action subsumes the second action. Since we assume that a more general action can perform all the tasks performed by its specialized actions, ACR metric assigns 1 as the action cover rate value for this case. This value shows that first action can fully provide all the functional capabilities of the second action in exactly same way. Second case is the reverse of the first case where the first action is subsumed by the second action. In this case the first action is a more specific action than the second action, therefore the first action can perform only some functional capabilities of the second action. To demonstrate this situation ACR metric assigns a value smaller than 1 as action cover rate value by using θ parameter. To explain these two cases consider `PayCash` and `PayCashDollar` actions in our ontology. We compute the action cover rate value of `PayCash` on `PayCashDollar` using the first case since `PayCash` subsumes `PayCashDollar`. This means `PayCash` can provide all the functional capabilities of `PayCashDollar`. On the other hand, we compute the action cover rate value of `PayCashDollar` on `PayCash` using the second case since `PayCashDollar` is subsumed by `PayCash`, which means `PayCashDollar` can provide only some of the functional capabilities of the `PayCash` action. As result ACR value of `PayCash` on `PayCashDollar` is higher than the action cover rate value of `PayCashDollar` on `PayCash`. In the third case, the considered actions are siblings. Although in this case there is no super or sub-concept relations between the two actions, since these actions have a common root they are still related to each other. However, in this case the resulting semantic similarity value must be smaller than the previous case, since the similarity between the concepts are weaker than previous cases. We achieve this effect by assigning γ a smaller value than θ. For instance, in our ontology `PayCashDollar` and `PayCashEuro` actions have a common root as `PayCash`, therefore both of them are specialized versions of `PayCash` action and share common functional capabilities, which make them similar in our context. In the last case, since there is no relation between the actions, ACR metric assigns 0 as semantic similarity value

between the actions. In our ontology `Pay` and `Search` are actions match to this fourth case. They have no common parent action (we ignore the meta-concept `thing`) and therefore they have no common functionality.

5. MODEL CHECKING BASED MATCHMAKING METHOD

In this chapter we present our second matchmaking method. In this method we use model checking techniques for matchmaking. We model services in Promela language and requests as set of LTL formulae. Our method uses Spin Model checker [22] with the information obtained from the service and request models for matchmaking.

5.1. Modeling Services and Requests for Model Checking

In this method we use a model checking took called Spin, which uses Promela language [22] to model systems. Hence we use Promela to model services. However, Promela provides us the same expressive power as an FSM to model a service, therefore it is simple to convert an FSM model into Promela or vice versa, without loosing any piece of information.

In our method we model consumer requests as a set of LTL formula, where each formula in the set represents a different property requested by the consumer. Similar representations are used in [23]. Let us explain what type of consumer properties we can formulate in LTL through some examples. These examples are on e-commerce domain, however the idea can be applied to any domain. In the following, in LTL formulae each letter represents a fact. For instance to state the fact that the payment is made for a good we write p, which can also be interpreted as $isPaid(payment, good)$. However, to simplify the formulae we use just letters and explain the meaning of each fact explicitly in the text.

Guarantee delivery or refund after payment: The service consumer requests a service that guarantees to deliver the ordered goods after the payment is done. In the exception case, if the payment is done but there is no delivery of the ordered item (due to the cancellation of the order by the consumer or a problem faced

by the provider, which prevent the provider from delivery of the ordered goods), then the payment must be refunded by the service provider. Following LTL formula corresponds to this situation, where p represents $isPaid(payment, good)$, d represents $isDelivered(good, customer)$ and r represents $isRefunded(p)$. Formally, $\mathsf{G}(p \rightarrow ((p\mathsf{U}d) \vee (p\mathsf{U}r)))$.

Expected final condition: This property is the expected final condition, where the world is either in a state so that the consumer made the payment and the provider delivered goods (successful transaction) or in a state, where there is no payment and delivery action performed by any of the participants (canceled transaction). In the formula p represents $isPaid(payment, good)$ and d represents $isDelivered(good, customer)$. Formally, $\mathsf{F}((p \wedge d) \vee (\neg p \wedge \neg d))$.

Delivery before payment: For some reason (i.e. the consumer is going to use the service for the first time and she does not trust the service) a consumer may request from a service that the delivery of the good is made before the payment for these good. This fact is represented by the following LTL formula. In the formula p represents the fact $isPaid(payment, good)$ and d represents $isDelivered(good, customer)$. Formally, $\mathsf{G}(d \rightarrow (d\mathsf{U}p))$.

Secure connection while doing payment: Following property represents the request that a secure connection must be established and the connection stay in this secure state until the payment operation completed. In other case the payment operation is not performed. In the formula s represents the fact $isSecure(connection)$ and p represents $isPaid(payment, good)$. Formally, $\mathsf{G}((s \rightarrow (s\mathsf{U}p)) \vee \neg p)$.

Secure connection for all the transactions: A more suspicious consumer who concerns more about her privacy may not feel comfortable with the previous property and require a service, where whole connection is secure. This property is represented by the following formula where s represents the fact $isSecure(connection)$. Formally, $\mathsf{G}(s)$.

In the real world, consumers may have different priorities for each of their requested properties according to their needs and expectations from services. For instance, for a consumer, who is concerned about her privacy, security properties of a service are essential. On the other hand, for a consumer, who wants to get her delivery as soon as possible, security may not be the main concern. To handle such situations our method provides a facility to consumers to assign one of two priority degree to their requested properties. We call these degrees as primary and secondary degrees, where properties with the primary degrees have higher priority than properties with the secondary degree. We use these degrees in our matchmaking method as explained in Section 5.2.

5.2. Model Checking Based Matchmaking Method

In our method, a consumer request consists of a set of LTL formula, where each LTL formula represents a different property that is required by the consumer from the service. Additionally each requested property is associated with a degree that shows the priority of the property from the consumers perspective. Considering these issues, we perform matchmaking in two steps. In the first step, which we call property matching explained in Section 5.2.1, by using model checking techniques, we check each requested property against all available services and accordingly assign a degree of satisfaction value in range [0, 1] between the property and each service, where higher degree of satisfaction values show that the property is better satisfied by the service comparing to other services with lower degree of satisfaction value. In the second step, which we call priority factoring explained in Section 5.2.2, using the priority values of the properties we combine the degree of satisfaction values between individual properties and services that we assigned in step one and compute an overall degree of satisfaction value between the request and each service. We use the overall degree of satisfaction values to sort and filter services according to the consumer needs.

5.2.1. Property Matching

In property matching, by using model checking techniques we check individually each property of a request against all available services and according to the result of the model checking we assign a degree of satisfaction value between the request property and the service. However, a model checker can only tell us, whether a property is satisfied or not. That is the result of the model checking is either true (property is satisfied by the service) or false (property is not satisfied by the service). This binary result of model checking restricts us only to services that exactly satisfy all the properties and prevents us from finding alternative services that can satisfy the consumer requirements partially. For instance, using binary results of model checking, we cannot capture the partial match between a request that requires a credit card payment facility and a service that provides a cash payment facility. Although the service does not match exactly to the request, there is still a degree of satisfaction between them and in the domain of service matchmaking capturing such partial matches is a required behavior of a matchmaking method.

To handle this issue, we propose an algorithm where to find alternative services we generate alternative request properties from the original request property using the relations defined in an action ontology. We compute the similarity between each generated alternative property and the original property using our ACR metric and associate this similarity with the alternative property. As the last step we select the most similar alternative property that is satisfied by the service and use its similarity value as the degree of satisfaction value between the original property and the service.

In this way we try to determine the degree of satisfaction of the service to the original request property by measuring the similarity between the original property and the alternative property, which is generated from the original property and can be satisfied by the service. Additionally, since we can measure the degree of satisfaction between the original and alternative property we can obtain a numeric measure to compare the services according to their degree of match.

Require: Service *serv*
Require: Property *prop*
Require: Ontology *onto*
1: **if** $serv \models prop$ **then**
2: return 1.0
3: **else**
4: $altPropSet = \text{genAltPropSet}(onto, serv, prop)$
5: $highSim = 0.0$
6: **for** *altProp* in *altPropSet* **do**
7: **if** $s \models alternativeProperty$ **then**
8: $altSim = \text{compSim}(prop, altProp, onto)$
9: **if** $altSim > highSim$ **then**
10: $highSim = altSim$
11: **end if**
12: **end if**
13: **end for**
14: return *highSim*
15: **end if**

Figure 5.1. Flexible matchmaking algorithm

Algorithm 5.1 explains our property matching method for one request property and one service. In the algorithm first by using the model checker we check whether the property is satisfied by the service or not (line 1). If the property is satisfied by the service, the degree of satisfaction value is set to 1, which shows an exact match, and no further computation is necessary (line 2). If the property is not satisfied as it is, we check whether the service satisfies alternative properties that are similar to the original required property. Therefore, we query the action ontology to find semantic relations between the actions in the service and the requested property (Algorithm 5.2) and use these relations to generate alternative properties that are similar to the original property (Algorithm 5.3) (line 4). Next, again using the model checker, we test each of these alternative properties against the service (line 6). If an alternative property is satisfied by the service, we compute the similarity of the alternative property to the

```
Require: Service serv
Require: Property prop
Require: Ontology onto
1: for propProc in prop do
2:     for servProc in serv do
3:         if semSim(propProc, servProc) > 0 then
4:             relDict[propProc]+ = servProc
5:         end if
6:     end for
7: end for
8: enumAltProp(prop, relDict, altPropSet)
9: return altPropSet
```

Figure 5.2. Generation of alternative properties.

original property by using the ACR metric (line 8). This value is in range (0, 1), which shows the property is partially satisfied. After all alternative properties are checked, we determine the alternative property with the maximum similarity to the original property and associate this similarity value as the degree of satisfaction value of the service for the considered original property (lines 9, 10).

As explained above, Algorithm 5.1 relies on Algorithm 5.2 to find related actions between a request and a service and on Algorithm 5.3 to generate alternative properties that are similar to the original property. Let us visit these algorithms next.

Algorithm 5.2 finds semantic relations between the actions in the required property and the actions in the service. For example, if both the request and the service contain an action related to payment, then payment is an action that is returned by Algorithm 5.2. To do this, the algorithm checks each action in the required property against each action in the service for a semantic relation using the action ontology. If there is a semantic relation between these two actions (line 3), then the algorithm stores the relation in a dictionary structure for future use (line 4). At the end of this process, for each action of the required property, the dictionary holds a set of actions,

Require: Property *prop*
Require: RelationDictionary *relDict*
Require: alternativePropertySet *altPropSet*
1: **if** all *proc* of *prop* considered **then**
2: *altPropSet+ = genAlt*
3: **else**
4: **for** *rel* in *relDict* **do**
5: *genalt+ = proc*
6: enumAltProp(*prop*, *relDict*, *altPropSet*)
7: **end for**
8: **end if**

Figure 5.3. Enumeration of alternative properties.

which are semantically related to the action and are contained by the current service.

Algorithm 5.3 generates alternative properties using the original property and the dictionary of relations created in Algorithm 5.2. It first enumerates recursively all possible combinations of the relations in the dictionary of relations (line 4) and then creates a new alternative property for each enumerated combination by replacing the actions in the original property with the actions in the enumeration (line 6).

Let us walk through the algorithms with an example. Assume that we have a service, where the first action is ordering a book (`#order`), the next action is the delivery of the book by mail (`#deliver_by_mail`) and the last action is paying for the book with cash (`#pay_by_cash`). The consumer looks for a book selling service and has the required property of delivery of the book before the payment. The consumer also wants to get the delivery by cargo (`#deliver_by_cargo`) and makes the payment with credit card (`#pay_by_creditcard`). We also assume that in our action ontology, we have relations between `#deliver_by_mail` and `#deliver_by_cargo`, and between `#pay_by_cash` and `#pay_by_creditcard` actions.

Algorithm 5.1 starts by checking if the property can be satisfied by the service

Table 5.1. Alternative properties generated from the original property

Req. Property	#pay_by_creditcard	#deliver_by_cargo
Alt. 1	#pay_by_creditcard	#deliver_by_mail
Alt. 2	#pay_by_cash	#deliver_by_cargo
Alt. 3	#pay_by_cash	#deliver_by_mail

as it is (i.e., if there is an exact match). Since the delivery and payment actions are different in the service and in the requested property, this check fails. Therefore, we need to check whether the service can partially satisfy the property or not. To do this, Algorithm 5.2 determines the relations between the actions in the required property and the service and creates the dictionary to hold the relations. The dictionary will contain the following:

- property.#pay_by_creditcard ↔ service.#pay_by_cash
- property.#deliver_by_cargo ↔ service.#deliver_by_mail

Next, Algorithm 5.3 creates alternative properties from the original property by enumerating all combinations in the dictionary. The alternative properties created in our example is listed in Table 5.1. After the alternative properties are generated, Algorithm 5.1 continues by checking each alternative property against the service. If an alternative property is satisfied by the service, it computes the similarity between the alternative and original property based on the average similarity of the individual actions in the original and alternative properties. For instance, if we consider the alternative property 3 in Table 5.1, the actions #pay_by_creditcard and #deliver_by_cargo are replaced with #pay_by_cash and #deliver_by_mail respectively. If we assume that the similarity computed between #pay_by_creditcard and #pay_by_cash is 0.8 and between #deliver_by_cargo and #deliver_by_mail is 0.6, then the overall similarity of the alternative property 3 to the original property is the average of these two values: 0.7. As the last step, the algorithm determines the alternative property with the maximum similarity value and returns this similarity value as the degree of satisfaction to the original property for property matching.

5.2.2. Priority Factoring

After the degree of satisfaction values are computed in property matching, we combine these values according to property priorities defined by the consumer and reach a final degree of satisfaction value at the request level.

Our method allows definition of two degrees of priority for each property. The first degree of properties are primary properties, which must be satisfied by the service. That is, the degree of satisfaction computed for this property in the property level must be greater than 0. Otherwise the matchmaker does not match the service to the request. For instance a consumer may want to guarantee that a secure connection is established before a payment transaction and that this connection stays secure until the transaction is completed. In such a case this property must be defined as a primary property and only the services that satisfy this property are returned by the matchmaker.

Properties in the second degree are called the secondary properties. These properties are recommended by the consumer, however the matchmaker may still match a request to a service which does not satisfy these recommended secondary properties. An example for such a secondary property might be related to the order of the actions. For instance, in a service to buy books, the consumer may prefer to get the books before she makes a payment. However the consumer may still accept services where payment is done before delivery if there is no better alternative and since the service satisfies the primary properties such as buying a book.

For services that satisfy all the primary properties, we compute the degree of satisfaction value in the request level as the linear sum of the individual degree of satisfaction values computed in the property level. For instance if the request has one primary property with degree of satisfaction value 0.8 for a corresponding service and one secondary property with degree of satisfaction value 0.6 for the same corresponding service than the overall degree of satisfaction value of the service for the property is 0.7. This scheme does not consider any importance between the primary and secondary properties, except that the primary properties must be satisfied by the service but the

secondary properties are not. If we want to emphasize primary properties further we might weight property priorities or we might weight each property individually to emphasize importance of some properties for the consumer. Using any of these schemes as the result of matchmaking process we will obtain a set of matching services where each service is associated with a degree of satisfaction.

5.3. Implementation of the Proposed Matchmaking Method

In our implementation of the model checking based matchmaking method, we use Spin as the model checker and implement our approach on top of it. Spin requires Promela language to model the services, but Promela models are not supported by any existing work on Web services. To fill this gap Ankolekar *et al.* [23] propose a mapping between Promela models and process model component of OWL-S. In our study we use this mapping to make conversions between Promela and OWL-S process models of services. Additionally, in order to use OWL-S also for service requests, we need a formalism to represent LTL formula in OWL-S, since we use LTL formulae to represent requested properties of users. In order to achieve this goal we extend Semantic Web Rule Language for First Order Logic (SWRL-FOL) [24], a version of Semantic Web Rule Language (SWRL) [25] with first order logic support, in such a way that it supports LTL formulae. We explain the details of this extension in the rest of this section.

OWL-S allows SWRL formulae to be embedded into OWL-S descriptions through its `Expression` class. However, since SWRL does not support LTL implicitly, we need to extend SWRL in order to embed LTL formulae into OWL-S. Since SWRL supports only Horn like rules, it is not capable to represent every LTL formula. However the SWRL-FOL extension of SWRL allows expression of First Order Logic (FOL). Hence we extend SWRL-FOL with LTL connectives U, G, F and X. In this way we can embed LTL formulae into OWL-S and describe service requests in it. The abstract syntax of our extension is listed in Figure 5.4. To make the use of this extension clear we provide an example XML code in Figure 5.5, which expresses the LTL formula $\mathsf{G}(p \rightarrow ((p\mathsf{U}d))$, where p stands for payment and d stands for delivery.

```
axiom := assertion

assertion := 'Assertion('
    [URIreference] {annotation} formula {foformula}
    ')'

foformula := atom
    | 'Until(' foformula foformula ')'
    | 'Globally(' foformula ')'
    | 'Finally(' foformula ')'
    | 'Next(' foformula ')'
    | 'And(' {foformula} ')'
    | 'Or(' {foformula} ')'
    | 'Neg(' foformula ')'
    | 'Implies(' foformula foformula ')'
    | 'Equivalent(' foformula foformula ')'
    | 'Forall(' variable {variable} foformula ')'
    | 'Exist(' variable {variable} foformula ')'

variable := 'I-variable(' URIreference description ')'
    | 'D-variable(' URIreference dataRange ')'
```

Figure 5.4. Abstract syntax of SWRL-FOL extension

```
<Assertion owlx:name="SWRL-LTL Example">
 <owlx:Annotation>
  <owlx:Label>SWRL-LTL rule example</owlx:Label>
 </owlx:Annotation>
 <Globally>
  <ruleml:Var type="xsd:boolean">payment</ruleml:Var>
  <ruleml:Var type="xsd:boolean">delivery</ruleml:Var>
  <Implies>
   <swrlx:classAtom>
    <owlx:Class owlx:name="isPaid"/>
    <ruleml:var>payment</ruleml:var>
   </swrlx:classAtom>
   <Until>
    <swrlx:classAtom>
     <owlx:Class owlx:name="isPaid"/>
     <ruleml:var>payment</ruleml:var>
    </swrlx:classAtom>
    <swrlx:classAtom>
     <owlx:Class owlx:name="isDelivered"/>
     <ruleml:var>delivery</ruleml:var>
    </swrlx:classAtom>
   </Until>
  </Implies>
 </Globally>
</Assertion>
```

Figure 5.5. SWRL-LTL example

6. THE MATCHMAKING FRAMEWORK

To realize the matchmaking methods that we propose in the previous chapters, we developed a service matchmaking framework, which provides an abstract architectural structure. Our framework has a modular structure. This modular structure has two main advantages. First, it allows us to use multiple matchmaking methods in conjunction. Second, it allows us to add, modify and remove matchmaking methods in the system in an ad-hoc manner without disturbing the overall structure. It is also important to note that this framework is not limited to realize only our proposed methods but it can be used to realize any matchmaking method that satisfy the requirements of the framework.

Figure 6.1 presents an overall picture of our framework. In our framework there are three types of entities. These are service providers, service consumers and the matchmaker agent. There is no restriction for the implementation of service providers and service consumers, except that they have to have capabilities to communicate with the matchmaker agent in a common protocol. Therefore we do not discuss the implementation of service providers and service consumers here. Instead we specify a simple protocol to show the communication requirements between the matchmaker agent and these entities. However the matchmaking agent should follow some specifications. We discuss these specifications in detail.

6.1. A Simple Protocol to Communicate with the Matchmaker Agent

In our framework an entity can communicate with the matchmaker agent in one of the two roles. That is, the entity can act either as a service provider or as a service consumer. In both cases there are a couple of messages that can be sent or received to communicate with the matchmaker agent. We will specify the messages for both cases in the rest of this section. It is important to note that we ignore the exception and error messages for the simplicity of our presentation.

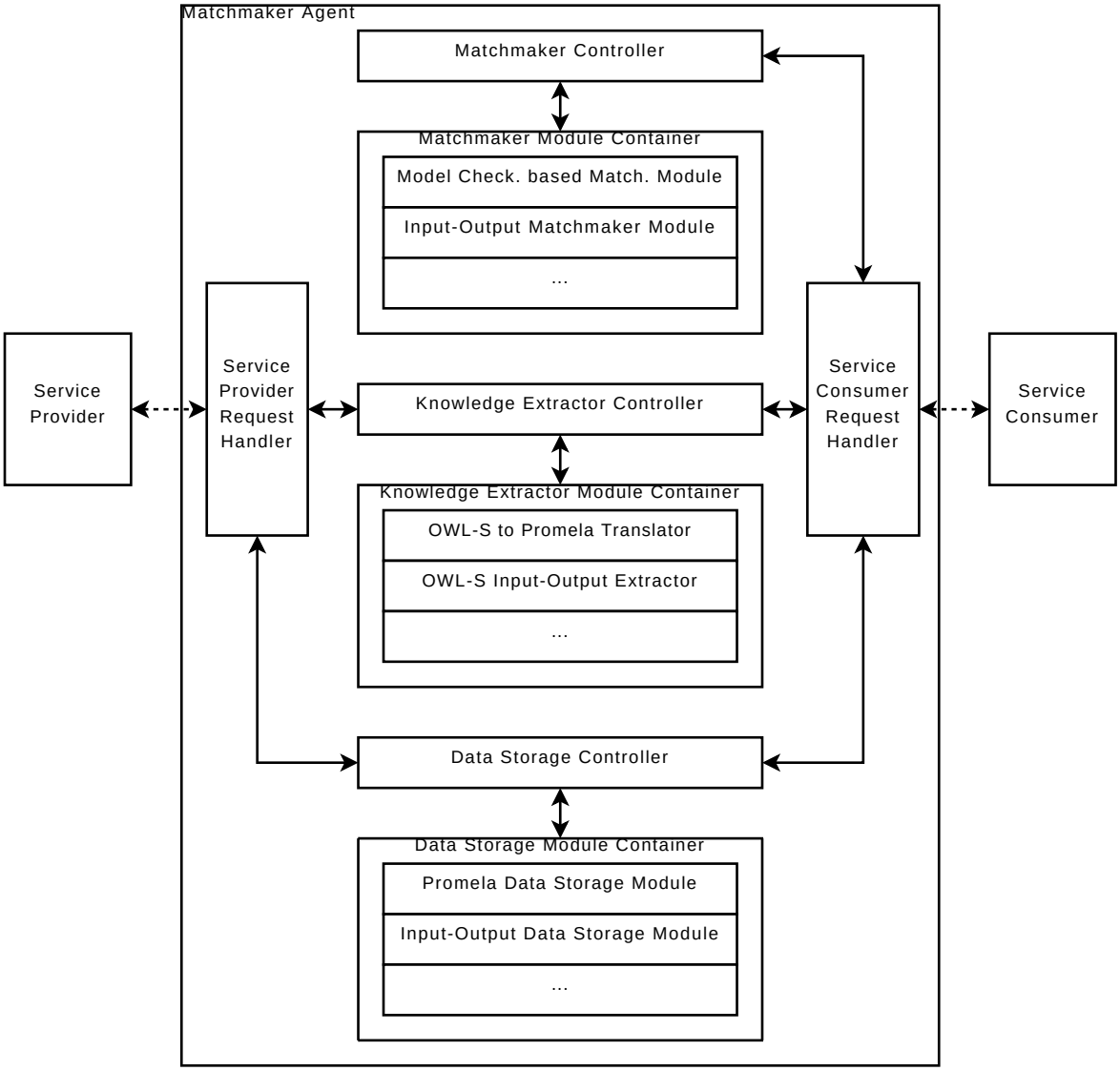

Figure 6.1. The matchmaking framework

6.1.1. Communicating as Service Provider

The aim of the service provider is to register its services to the matchmaker agent for advertisement. In order to do this the service provider must use the protocol we present in Figure 6.2. In this protocol, the service provider sends first a `new_serv_reg_req` to request from the matchmaker agent to register the new service to its database. If everything is well, the matchmaker agent replies with the message `new_serv_reg_rep` and sends a unique id number associated with the new service. After the reply message received, the service provider should send one or more service descriptions to the service matchmaker depending on the requirements of the existing

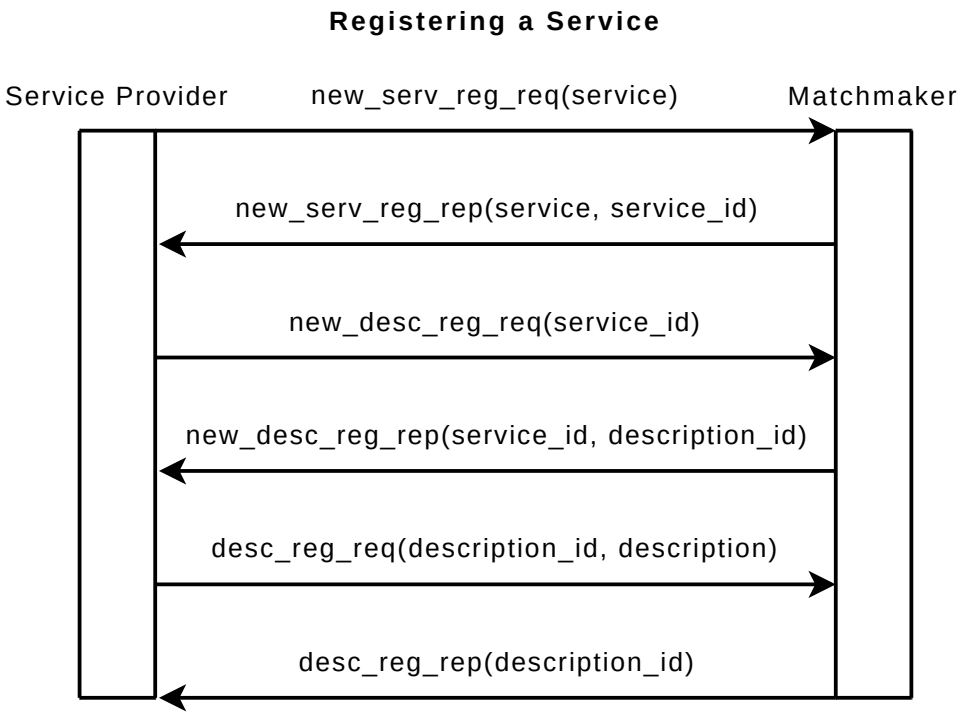

Figure 6.2. Service register protocol

matchmaking methods involved by the matchmaker agent. To send a service description, the service provider first sends the message called `new_desc_reg_req`. This message consists of the id that is provided by the matchmaker in the previous message, so that the matchmaker can associate the new service description with the registered service. The matchmaker agent replies with `new_desc_reg_rep` message, which involves a unique id number for the new description. Then using this description id number, service provider sends the `desc_reg_req` message, which involves the service description itself. The matchmaker agent replies with a `desc_reg_rep` message if everything goes well. The service description registry sequence can be repeated several times by the service provider to describe the same service through different service description languages for different matchmaking methods.

6.1.2. Communicating as Service Consumer

The aim of the service consumer is to find a service to satisfy its needs. To find this service it makes a service request from the matchmaker agent using the protocol we present in Figure 6.3. To initiate the protocol, service consumer sends `serv_req_req`

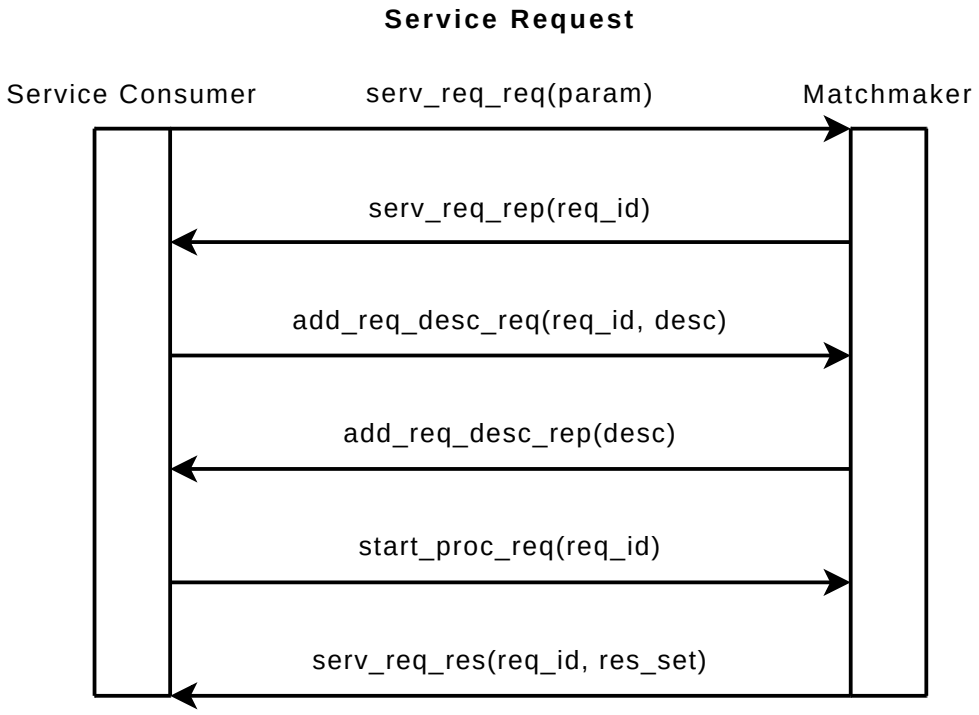

Figure 6.3. Service request protocol

message. With this message, service consumer can specify optional parameters like which matchmaking methods are going to be used for this request and the weight of each method, which is going to be used to combine the results of the individual matchmaking methods that are used by the matchmaker agent. If no method is specified in the request, the choice of the method is left to the matchmaker agent. Similarly if no weights are specified for the methods, weight values are left to the choice of the matchmaker agent. As reply to the `serv_req_req` message, the service consumer receives the `serv_req_rep` message form the matchmaking agent with a unique id to identify its request. Then using this id, service consumer start to send `add_req_desc_req` messages, which describe the requested service by the service consumer. The service consumer can send several `add_req_desc_req` messages according to the requirements of the matchmaking methods. For each `add_serv_desc_req` message the service consumer also receives a `add_serv_desc_rep` message sent by the matchmaker agent to notify the new request description is received. After the service consumer sent all `add_req_desc_req` messages, it sends a `start_proc_req` message to indicate the matchmaker agent to start processing the service request. After the request is processed, the matchmaker agent returns a list of services that satisfy the request and a numeric value for each ser-

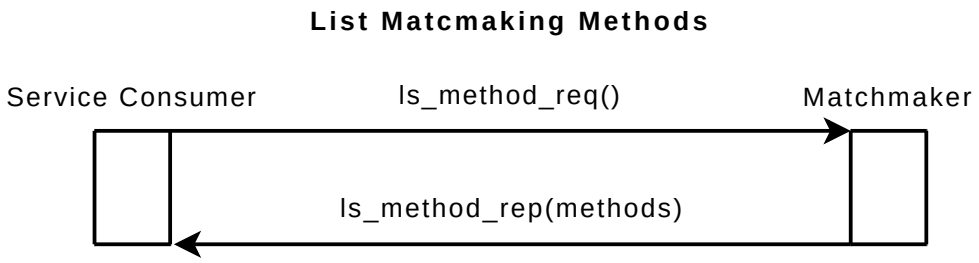

Figure 6.4. Matchmaking method listing protocol

vice in the list that shows the degree of satisfaction for the associated service through a `serv_req_res` message.

6.1.3. Obtaining a List of Matchmaking Methods

In some cases for both service providers and service consumers it is necessary to know the matchmaking methods that the matchmaker can use. For this purpose there is a simple method listing protocol we present in Figure 6.4, which can be used by both service providers and service consumers. To initiate this protocol, the interested entity sends the message `ls_method_req` to the matchmaker agent. The reply of this message involves a list of matchmaking methods and their definitions. The definitions can be in any format, which is left to the matchmaker agent. If the interested entity is a service provider, it can use the list to determine which type of service descriptions it should send while registering a new service or to update information about its already registered services. On the other hand a service consumer can use this list to select the methods and their weights that are going to be used by the matchmaker agent, when a service request is made by the service consumer.

6.2. The Matchmaking Agent

In the framework, the matchmaker agent acts as a middle agent [13] and matches service requests of the service consumers to the service advertisements of the service providers. The most important aspect of the matchmaker agent is its modular structure, which allows the use of multiple matchmaking methods in conjunction. Through this modularity, for a service consumer it is possible to specify in a service request,

which matchmaking method(s) are going to be used by the matchmaker for this service request. This allows the consumers to control properties of the matchmaking process like performance and quality according to their needs. The modular structure also allows us to add, modify and remove matchmaking methods in the matchmaker agent in an ad-hoc manner without interrupting the operation of the agent. The matchmaker agent is composed of four types of entities. These are *task modules*, *module containers*, *module controllers*, and *request handlers*.

6.2.1. Task Modules

Task modules or simply modules are the primitive entities that are responsible for performing specific data processing tasks. There are three types of modules: *knowledge extractor modules*, *data storage modules* and *matchmaker modules*.

Knowledge extractor modules: These modules are responsible for extracting the information required by matchmaking methods from the service and request descriptions. For instance in our first method service providers and consumers use the process model of OWL-S to represent FSM models of services. However, our implementation of this method uses an internal data structure to represent the service model and therefore we have to map the process model of OWL-S to this internal data structure. In the framework we have a knowledge extractor module that is responsible for this mapping. Similarly, in our second method we need Promela models of services to use Spin model checker for matchmaking. However, service providers register their services through OWL-S, which does not include the Promela model of the service. Therefore, we have a knowledge extractor module in the framework to generate the Promela model of the service by using the information extracted from the OWL-S service model. In both cases if another language is required to describe services, we can simply add a new module that is capable of performing the same extraction task for the new language, which makes the matchmaker agent capable of working with the new language without

any modification to the rest of the system.

Data storage modules: These modules are responsible from the storage of the knowledge that is extracted by the knowledge extractor modules from service descriptions. Each matchmaking algorithm may require different type of knowledge to work. For instance our first method uses an internal data structure to represent the FSM models of services. On the other hand, our second method requires the model of the service in Promela and another method may require the some other knowledge like input-output interface of the service. By separating the data storage modules of each method we preserve flexibility to modify our overall system in an ad-hoc manner without disturbing the other components of the system. Additionally, this method distributes the data over several modules, which allows us distributed processing and increase scalability and fault tolerance of the system.

Matchmaker modules: Matchmaking modules are the implementations of the actual matchmaking algorithms. For instance each of our FSM and model checking based matchmaking algorithms are implemented as separate matchmaking modules.

6.2.2. Module Containers and Controllers

Module containers provide an abstraction layer to collect task modules together according to their module types of knowledge extractor modules, data storage modules, and matchmaker modules. Module controllers provide an interface to communicate with the modules of the corresponding container. Only controllers know the modules that are contained by the corresponding container and how to communicate and use these modules. Therefore the controller is responsible to find the appropriate module for a specific task.

6.2.3. Request Handlers

Request handlers handle requests from the service providers and service requesters as well as manage the data flow between the modules through corresponding controllers. There are two different handlers. *Service consumer request handler* is responsible for handling service matchmaking requests from service consumers. It collects information from knowledge extractor and data storage controllers and feed the matchmaker controller with this information and sends the matchmaking results back to the service consumer. *Service provider request handler* is responsible for handling service registry requests from service providers. It requests all the required service information for an incoming registry request from the knowledge extractor controller and forwards this information to the data storage controller to finalize the registration.

6.3. Implementation Scenario for the Framework

In this section we will discuss over an implementation scenario to investigate our framework in detail. In this scenario there are three matchmaking methods. Two of these matchmaking methods are our proposed methods. The other matchmaking method is the input-output based matchmaking method, which is widely investigated in the literature [10, 11, 12]. We include this third method in this scenario to show that our framework is also suitable for matchmaking methods other than we propose in this thesis. First, we will discuss how to add new matchmaking methods to the matchmaker agent in our framework. Then we will discuss what happens when a request from a service provider is received by the matchmaker agent to register a new service to the system and lastly we will discuss what happens when a service request from a service consumer is received by the matchmaker agent.

6.3.1. Matchmaking Methods

A new matchmaking method can be added to the system by simply adding the modules related to the new matchmaking method to the corresponding module containers and by informing module containers about the new modules.

To add a new matchmaking method the definite module that must be added to the framework is the matchmaker module, which involves the implementation of the matchmaking method. The matchmaker controller must also be informed about the new matchmaking methods in order to use them. Other modules (knowledge extractors and data storage modules) are optional to add a new matchmaking method to the system and may be ignored in some situations. For instance, if the existing knowledge extractor modules in the system can provide enough information for the new matchmaking method, then the existing knowledge extractor module can be used also for the new matchmaking method and there is no need to add a new knowledge extractor module to the system. Similarly, if the existing data storage modules are capable to hold the required information for the new matchmaking method, than no new data storage module is necessary. However, if any of the existing modules cannot satisfy the needs of the new matchmaking method, than it is a must to add the required modules to the system in order to use the new matchmaking method properly. One important point to note here is in both cases the knowledge extractor and data storage controller must be informed about which modules are going to be used with the new matchmaking method.

In our scenario we add three new matchmaking methods, namely the FSM based, model checking based and input-output based methods, to our framework. Each of these matchmaking methods requires different type of knowledge, therefore we have to add new knowledge extractor and data storage modules for each of them. For instance, for the FSM based method, we need a knowledge extractor that can extract the necessary information from the process model of the OWL-S service description and an data storage module that can store the extracted knowledge. In the case of model checking based matchmaking method, we need two knowledge extractor modules. One module to extract the service model represented by the process model of OWL-S and convert it into Promela language in the case of service registry and one module to extract temporal logic properties from the extended OWL-S description in the case of a service request. However we need only one data storage module to store converted Promela models, since in the case of a service request the extracted temporal logic properties are only used but not stored by the matchmaker agent. In the case of

input-output matching, one knowledge module is required, that extracts the input and output parameters from OWL-S service descriptions and one data storage module to store these extracted parameters.

This scenario shows us the modular structure of our matchmaking framework allows us to add new matchmaking methods in an ad-hoc manner without disturbing existing matchmaking methods. Modification and deletion of existing matchmaking methods work in the same way of addition. For instance, when we decide to use not OWL-S but another description language to model services for model checking based method, we only need to add a new knowledge extractor module that is capable of converting the service model from this new description language to Promela. This will neither affect the other modules related to the model checking based method nor modules related to the other matchmaking methods in the framework. This situation holds also for the deletion case.

6.3.2. Registering new Services

In our matchmaking framework, service providers register their services to the matchmaker agent in order to advertise them to service consumers. To do this, service providers create service descriptions and send them to the matchmaker agent. Depending on the existing matchmaking methods and related modules, service providers may create more than one service description for the same service. In our scenario, all existing matchmaking methods use OWL-S for service descriptions. Therefore, it is enough for a service provider to send only one description for each service.

The matchmaker agent handles a service register request of a service provider as follows. Following the protocol that we specify in Section [REF], the service provider sends first a service register request. This request is handled by the service provider request handler of the matchmaker agent. The handler first generates a unique id for the new service and sends this id back to the service provider. After the reply of the matchmaker agent is processed, the service provider starts to send as many service description as required by the available service matchmaking methods. We show how

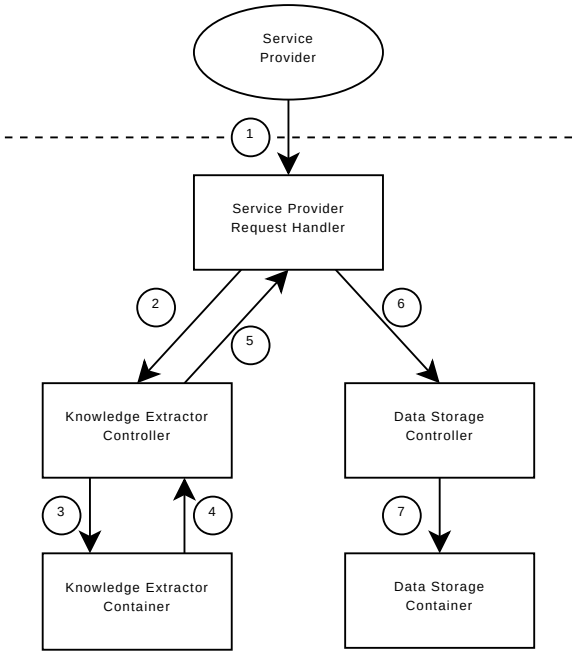

Figure 6.5. Flow of service registry process in the matchmaker agent

these service descriptions are processed by the matchmaker agent in Figure 6.5, where the numbers in circles show the order of flow in this process. These descriptions are handled by the service provider request handler, when they received from the service provider (1). The service provider request handler sends the service descriptions first to the knowledge extractor controller (2). According to the service description, the knowledge extractor controller calls appropriate knowledge extractor modules in the knowledge extractor module container (3). The knowledge extractor modules process the service descriptions and returns the extracted knowledge back to the knowledge extractor controller (4). The controller sends this information back to the request handler (5). The request handler associates the extracted knowledge with the corresponding matchmaking methods and directs the knowledge to the data storage controller (6). The data storage controller calls the appropriate data storage modules from the data storage module container in order to store this knowledge (7). Data storage controller uses the matchmaking method information associated with the received knowledge in order to determine which data storage module to call .

In our scenario it is enough for a service provider to send one OWL-S file to describe its registering service, since all the information required by the three match-

making methods can be obtained form a single OWL-S description. For FSM based method the process model of OWL-S is used by the corresponding knowledge extractor module to extract the required information by the matchmaking method. Similarly, for model checking based method, the corresponding knowledge extractor module also uses the process model of OWL-S and translate this process model into Promela language. For the input-output based method, the corresponding module extracts the input and output parameters of the service from the profile component of the OWL-S service description.

6.3.3. Making Service Requests

In our framework, service consumers ask for services to the matchmaker agent through sending service requests. A service consumer has two options related to the matchmaking operation and the algorithms while making a service request. The first option is to leave the details of the matchmaking operation and algorithms to the matchmaker agent. In this case the service consumer does not care about the details of the matchmaking operation. It simply defines the required properties from the potential service in a description language and left the rest to the matchmaker agent. The matchmaker agent tries to obtain as much information as it can get from the request of the service consumer and selects applicable matchmaking methods to maximize the matchmaking quality. Although this option simplifies the effort of the service consumer, since the service request created by the service consumer is too generic, the matchmaking methods may not get enough information to work properly and the matchmaking results may be poor. The second option for the service consumer is to tell the matchmaker agent which matchmaking methods to use for the service request and to provide all the required information by this matchmaking methods adequately. In this case the service consumer should have enough knowledge about the matchmaking methods and the information required by them. However, using this option the matchmaking quality increases significantly, since all the information required by the matchmaking methods are provided by the service consumer properly.

We show how service requests are processed by the matchmaker agent in Fig-

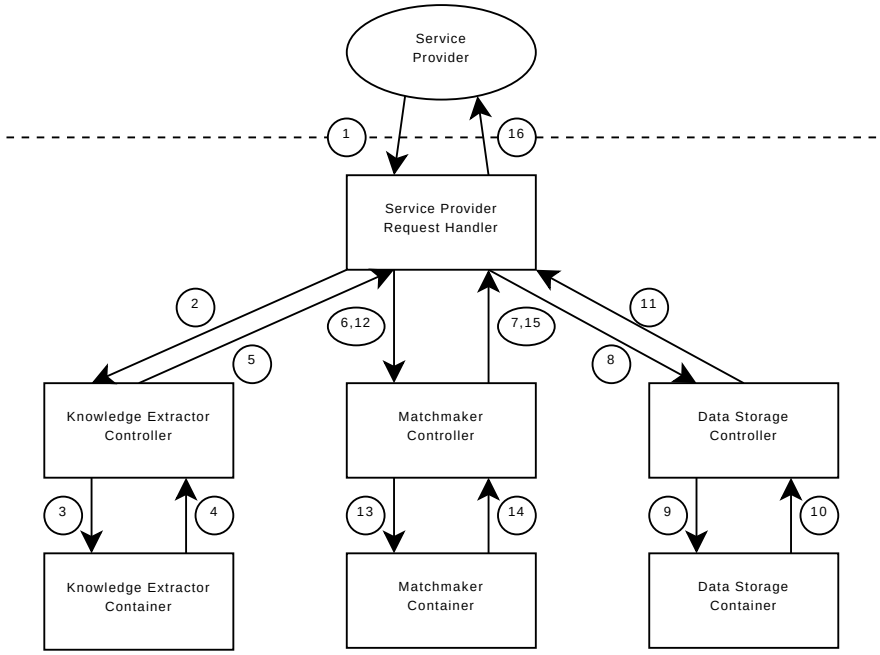

Figure 6.6. Flow of service request process in the matchmaker agent

ure 6.6, where the numbers in circles show the order of flow in this process. When a service consumer makes a service request, the service consumer request handler of the matchmaker agent handles this request (1). It firstly sends the service request to the knowledge extractor controller (2), which calls the knowledge extractor modules from the knowledge extractor module container that can interpret and extract the required knowledge from the service request (3). Knowledge extractor modules send the extracted knowledge back to the knowledge extractor controller (4). This extracted knowledge is sent back to the request handler by the knowledge extractor controller (5). Then the request handler sends the extracted knowledge to the matchmaker controller (6). Additionally, according to the request of the user, the request handler sends the names of the matchmaking methods that are going to be used for matchmaking to the matchmaker controller. After receiving this information the matchmaker controller ask to the request handler for the available service information that are related to the matchmaking methods that are going to be used (7). The request handler communicates with the data storage controller and requests this information (8). To get this information the data storage controller calls the necessary data storage modules from the data storage module container (9). Data storage modules retrieve this information and send it back to the data storage controller (10). Then the data storage con-

troller sends this information back to the service consumer request handler (11), which transfers this information to the matchmaker controller (12). Then the matchmaker controller calls the matchmaker modules from the matchmaker module container, as specified by the request handler at the beginning, with the collected information (13). The matchmaker modules performs the matchmaking operation and send the results back to the matchmaker controller (14). The matchmaker controller forwards these results to the service consumer request handler (15). The request handler combines the results according some mechanism and sends a list of services to the service consumer with their relevance to the request of the service consumer (16).

The mechanism to combine separate result of the different matchmaking modules used by the request handler may vary according to the situation under consideration. For instance, a simple mechanism can take the average of relevance value returned by each matchmaker module to derive a final relevance value for each service. Alternatives, a more specific method can use different weights for the results of different matchmaking modules. This mechanism is independent of the rest of the matchmaking framework, therefore any mechanism can be used here according to the specific use case.

In our scenario it is enough for a service consumer to create a single OWL-S file in order to make a service request. For FSM based method the process model of OWL-S is used by the corresponding knowledge extractor module to extract the required information by the matchmaking method. In the case of model checking based method, the corresponding knowledge extractor module extracts the LTL formulae from the extended SWRL expressions defined in Section 5.3. For the input-output based method, the corresponding module extracts the input and output parameters of the service from the profile component of the OWL-S service description.

7. CASE STUDY

We evaluate our proposed matchmaking methods through case studies. In the first part we evaluate our FSM and model checking based matchmaking methods separately. In the second part we compare the two methods and discuss their strengths and weaknesses.

7.1. Case Study of FSM Based Matchmaking Method

Our main aim in this case study to understand and evaluate the performance of the structural similarity metrics that we use in our FSM based matchmaking method. Additionally to examine the effect of the proposed semantic metric ACR we use it with all four structural similarity metric and present the results. In this case study we present one service request and six related services, which satisfy the request in different degrees.

7.1.1. Case Study Setup

In this case study, both the service request and the services consist of only one flow sequence. We ignore the case of multiple choices and loops in our case study, since these control structures are converted into single sequences during the matchmaking process and do not have any effect on the computation of the similarity values. In the rest of this section we verbally describe the service request and the six related services in detail. We also present the action sequences of the service request and the services to make the details more clear. In this case study we use a simple action ontology partially shown in Figure 3.1 to compute the ACR metric values.

Service Request: The service consumer is looking for a service to buy books. The user does not have any security concerns. Therefore she requests a non-secure connection. The service consumer requires the basic shopping capabilities from the service, like searching a book, adding the book into her cart, ordering the goods in her

cart and paying for her order. Specifically she wants to make the payment with a credit card. She also asks for an authentication mechanism to identify herself, however does not specify the mechanism. The action sequence of the service request is as follows; `#ConnectNonSecure` → `#SearchBook` → `#AddCartBook` → `#OrderCartBook` → `#Authenticate` → `#PayCreditCard`

Service-1: This service provides a cart mechanism for shopping books. However it is not possible to order and buy the books in the cart. To order and buy the books in the cart the service must be composed with another service that provides these functional capabilities. In general this service is a bad match for the service request, since the service does not allow the consumer to order and buy the books in the cart, which are the fundamental functional capabilities the service consumer requested. The details of the action flow for the service is as follows; The service first provides a facility to search for books. If the search result is successful, the user authenticates herself to the service and then can add the book found as the search result to her cart. The service provides a non-secure connection. The action sequence of the service is as follows: `#ConnectNonSecure` → `#SearchBook` → `#Authenticate` → `#AddCartBook`

Service-2: This service provides all the fundamental functional capabilities that are requested by the service consumer. However some of these functional capabilities are provided not by the actions given in the request but through similar actions. For instance the service consumer requests an authentication action in her request. However she does not require any specific mechanism and leave the choice to the service. To specify this issue the service consumer uses the action `#Authenticate`, which is the most general authentication action in the action ontology. Since the service provides the authentication by `#AuthenticateHTTPS`, which is a specialized version of the `#Authenticate` action, we can deduce that the service actually satisfies the authentication mechanism request of the service consumer. The two other actions that are different than the requested ones are the `#ConnectSecure` and the `PayMoneyTransfer` actions. However using the action ontology we can deduce that these actions partially provide the functional capabilities requested by the service consumer through

the actions `ConnectNonSecure` and `PayCreditCard` in the service request, respectively. Although the requested functional capabilities are not fully satisfied by this service, we can still consider it an average match, because of the relations in the ontology between the requested and provided actions. Action sequence of the service is as follows: `#ConnectSecure` → `#SearchBook` → `#AddCartBook` → `#OrderCartBook` → `#AuthenticateHTTPS` → `#PayMoneyTransfer`

Service-3: This service is to buy a DVD but not a book. Therefore this is not a good match for the request. However since the main purpose of the service is still to sell DVDs, it shares many common functional capabilities with a service to sell books, which is the source of the challenge for a matchmaking. The action flow of the service is as follows: `#ConnectNonSecure` → `#SearchDVDBasicTitle` → `#AddCartDVD` → `#OrderCartDVD` → `#Authenticate` → `#PayCreditCard`

Service-4: This service is a general service to buy any consumable item. Hence it also allows buying books. By checking the actions involved by the service using the action ontology we can see that the actions provided by the service are the generalizations of the actions required by the service consumer. Therefore the service can handle the functional capabilities required by the service consumer and hence this is a good match for the service request. Please note that the service also provides an action to cancel an order, which is not required by the service consumer. The action flow of the service is as follows: `#Connect` → `#Search` → `#AddCart` → `#OrderCart` → `#CancelOrder` → `#Authenticate` → `#PayCreditCard`

Service-5: This service provides more specific functional capabilities than the requested service. For instance it provides a book search facility, which accepts only a book title. This facility represented by the action `SearchBookBasicTitle`, which is a more specialized case of the action `BookSearch` expected in the service request. In general we can count this service as an average match to the service request, since it can provide the fundamental requested functional capabilities with some restrictions. The action flow of the service is as follows: `#ConnectNonSecure` → `#SearchBookBasicTitle`

→ #AddCartBook → #OrderCartBook → #AuthenticateSSH → #PayCreditCardVisa

Service-6: This service provides exactly the requested functional capabilities but through a different order of actions. The main difference is the place of the authentication operation, which is performed immediately after the connection is established. In the service request this operation is performed just before the payment action. This difference causes a shift in flow order of all actions. A good heuristic should recognize this shift and still match the service to the request. The flow of actions for the service is as follows: #ConnectNonSecure → #Authenticate → #SearchBook → #AddCartBook → #OrderCartBook → #PayCreditCard

In general, services 4 and 6 are the best matches for the service request. However services 2 and 5 can partially provide the requested functional capabilities. Therefore we expect that our matchmaking method should also match these two services. However the matchmaking method should emphasize that these two services are weaker matches than services 4 and 6. We also expect that the matchmaking method successfully determine that services 1 and 3 do not satisfy to the request.

7.1.2. Results

To measure the individual performance of each structural metric first we compute the similarity between the request and all services for each heuristic separately. Then we integrate the ACR metric to each structural metric and perform the same procedure to observe the effect of the use of semantic knowledge.

Table 7.1 presents our results. Each column shows the computed similarities between the request and each service using the associated metric. We add the letter S to the beginning of the metric names to indicate that they use ACR metric in addition to the structural similarity. In the ACR computations we take θ as 0.75 and γ as 0.5.

Considering these results, we observe the following:

Table 7.1. Matchmaking results of structural and semantic metrics

	CAC	SCAC	LCStr	SLCStr	LCSeq	SLCSeq	ED	SED
Serv-1	0.60	0.94	0.50	0.33	0.75	0.50	0.33	0.46
Serv-2	0.50	0.79	0.50	0.79	0.50	0.79	0.50	0.79
Serv-3	0.50	0.69	0.33	0.69	0.50	0.69	0.50	0.69
Serv-4	0.15	0.86	0.14	0.67	0.14	1.00	0.14	0.86
Serv-5	0.50	0.84	0.33	0.84	0.50	0.84	0.50	0.84
Serv-6	1.00	1.00	0.50	0.50	0.83	0.83	0.67	0.67

- CAC metric is particularly useful when the request and the service have the same functionality but different flows like in the case of Service-6. CAC metric can also successfully differentiate the unrelated services such as Service-3. However, it cannot detect that Service-1 is not a good match since CAC metric does not consider the difference between the number of actions in the request and the number of actions in the service.
- In general LCStr metric shows the worst performance compared to the other metrics, since it is strictly dependent on the order of the flow. It is only successful in the case of Service-1, where the service provides less functionality and therefore the longest common substring between the service and request is short.
- LCSeq metric is especially successful if the service covers the request and provides some additional functionality like in the case of Service-4. It can also successfully detect the two poor matches Service-1 and Service-3.
- ED metric can successfully differentiate between good and poor matches. The only exception occurs when flow orders are different like in the case of Service-6.
- We also observe that using the semantic similarity metric ACR with the structural similarity metrics we obtain more precise results compared to the use of structural similarity alone.

7.2. Case Study of Model Checking Based Matchmaking Method

In this case study we evaluate our model checking based matchmaking method. In this case study there is a user, who is looking for a service to buy books. She has a primary property and a secondary property in her service request. Primary property states that she requires a service where the connection is secured by https protocol when she makes the payment by her credit card and otherwise she will not do any payment. We represent this property with the LTL formula $\mathsf{G}((\texttt{secure-https-connection} \rightarrow (\texttt{secure-https-connection}\ \mathsf{U}\ \texttt{pay-by-creditcard})) \vee \neg\ \texttt{pay-by-creditcard})$. The secondary property states that she prefers to receive the book by cargo before she makes any payment by her credit card. We represent this property with the LTL formula G $(\texttt{delivery-by-cargo} \rightarrow ((\texttt{delivery-by-cargo}\ \mathsf{U}\ \texttt{pay-by-creditcard}))$. For each fact we use the shorthands as s_r for `secure-https-connection`, p_r for `pay-by-creditcard` and d_r for `delivery-by-cargo`.

There are 16 registered book selling services available in the matchmaker agent. These services differ in how well they satisfy the request. Broadly, there are five categories of services.

The first category contains services that exactly match the request by satisfying both of the requested properties. Service 16 is the only service in this category.

In the second category, there are services that satisfy the primary property exactly while satisfying the secondary property only partially or vice verse. Services 11, 12, 14 and 15 are on this category. Let us examine these services in detail. Services 11 and 12 satisfy the secondary property exactly. However these services can satisfy the primary property only partially. Service 11 offers alternative actions both for connection and payment methods requested in the original requested property. Service 12 can provide the credit card payment action, but the connection action is different than the originally requested one. On the other hand Services 14 and 15 exactly satisfy the requested primary property. However they can satisfy the secondary property only partially. Service 14 can provide only alternative actions for both payment and delivery actions.

Service 15 can satisfy the secondary property better, since it can at least provide the desired payment action. According to this information we obtain that service 15 is clearly the best one between the services in this category, since it can satisfy the primary property exactly and the secondary property with only one alternative action. Services 12 and 14 comes after service 15, however the decision about which one is better is not very clear. Service 12 can satisfy the primary property with one alternative action and the secondary property exactly. On the other hand service 14 can satisfy the primary property exactly, but to satisfy the secondary property with two alternative actions. Service 11 is definitely the worst one in this category.

In the third category there is only service 13, which satisfies the primary property exactly, but it can not satisfy the secondary property even with alternative actions. However, this service still matches to the service request, since the secondary properties are optional.

In the fourth category, there are services that satisfy the primary property partially and satisfy the secondary property either partially or not at all. Services 4, 5, 6, 7, 8 and 9 are the services in this category. Services in this category satisfy neither of the properties exactly, however they should still be considered since they might be useful for the service consumer with slight modifications in the service request. Service 9 can be considered as the best one in this category, since it needs only one alternative actions both to satisfy the primary and secondary properties. Services 8 is better than the others since it can satisfy the primary property with one alternative. From similar reason service 8 is better than service 7. Service 5 and 4 are the worst services, since they can not satisfy the secondary property in any case.

Finally, in the fifth category, we have services that do not satisfy the primary property. In this case without considering the secondary property, we ignore these services, in our method we expect that the primary property must be satisfied (at least partially with alternative actions) in order to be considered as a match to the service request. Service 1, 2, 3 and 10 are in this category.

Table 7.2. List of the 16 potential services

Service	Primary Property	Secondary Property
Serv-01	not satisfied	not satisfied
Serv-02	not satisfied	$\mathsf{G}(d_a \rightarrow (d_a \cup p_a))$
Serv-03	not satisfied	$\mathsf{G}(d_r \rightarrow (d_r \cup p_a))$
Serv-04	$\mathsf{G}((s_a \rightarrow (s_a \cup p_a)) \vee \neg p_a)$	not satisfied
Serv-05	$\mathsf{G}((s_r \rightarrow (s_r \cup p_a)) \vee \neg p_a)$	not satisfied
Serv-06	$\mathsf{G}((s_a \rightarrow (s_a \cup p_a)) \vee \neg p_a)$	$\mathsf{G}(d_a \rightarrow (d_a \cup p_a))$
Serv-07	$\mathsf{G}((s_a \rightarrow (s_a \cup p_a)) \vee \neg p_a)$	$\mathsf{G}(d_a \rightarrow (d_a \cup p_r))$
Serv-08	$\mathsf{G}((s_r \rightarrow (s_r \cup p_a)) \vee \neg p_a)$	$\mathsf{G}(d_a \rightarrow (d_a \cup p_a))$
Serv-09	$\mathsf{G}((s_a \rightarrow (s_a \cup p_r)) \vee \neg p_r)$	$\mathsf{G}(d_a \rightarrow (d_a \cup p_r))$
Serv-10	not satisfied	satisfied
Serv-11	$\mathsf{G}((s_a \rightarrow (s_a \cup p_a)) \vee \neg p_a)$	satisfied
Serv-12	$\mathsf{G}((s_a \rightarrow (s_a \cup p_r)) \vee \neg p_r)$	satisfied
Serv-13	satisfied	not satisfied
Serv-14	satisfied	$\mathsf{G}(d_a \rightarrow (d_a \cup p_a))$
Serv-15	satisfied	$\mathsf{G}(d_a \rightarrow (d_a \cup p_r))$
Serv-16	satisfied	satisfied

In general, we expect from the matchmaker to rank the services in such a way that, the services in the first category are on the top, then the services in the second category, then the services in the third category and lastly the services in the fourth category. However, services that are the best and worst of their categories may change their places with the top and worst services of following and leading categories, respectively. It is also expected that services in category five are going to be ignored totally.

The Table 7.2 lists all the services with their related properties to the request. If a property is exactly `satisfied`, we write satisfied in the corresponding field and if the property is not satisfied even when we use semantics, we write `not satisfied`. If the service does not satisfy the original property but satisfies a semantically similar property that is generated by our method, we present this alternative property in the

Table 7.3. Overall degree of match values

Service	Schema 1	Service	Schema 2
Serv-16	1.000	Serv-16	1.000
Serv-15	0.875	Serv-15	0.875
Serv-12	0.875	Serv-12	0.825
Serv-09	0.700	Serv-14	0.775
Serv-14	0.625	Serv-13	0.700
Serv-11	0.625	Serv-09	0.700
Serv-13	0.500	Serv-08	0.600
Serv-08	0.500	Serv-11	0.525
Serv-07	0.500	Serv-05	0.525
Serv-05	0.375	Serv-07	0.400
Serv-06	0.250	Serv-06	0.300
Serv-04	0.125	Serv-04	0.225
Serv-10	0.000	Serv-10	0.000
Serv-03	0.000	Serv-03	0.000
Serv-02	0.000	Serv-02	0.000
Serv-01	0.000	Serv-01	0.000

corresponding field. Facts s_a, p_a and d_a are alternative actions provided by the services that are related to s_r, p_r and d_r, respectively.

For simplicity, in our case study we take the similarity value between all semantically related actions as 0.5. This means, similarity between action s_r and s_a, p_r and p_a, and d_r and d_a are all equal to 0.5. According to this setting, the similarities between the original property requested by the user and the alternative properties satisfied by the services are computed by taking the average of the similarities between the individual actions in the properties. For instance, in the original primary property, there are s_r and p_r actions. Assume that there is an alternative property where s_r action is identical but instead of p_r, p_a is used. The similarity of the properties will be equal to 0.75, which is the average of 1.0 from s_r and 0.5 from p_a.

We also introduce two different weight schema between primary and secondary properties in the computation of the overall degree of match for a service to show the influence of the weights over the result quality. In the first schema both primary and secondary properties have equal weight as 0.5 each. In this schema we do not favor any of the properties. In the second schema, the primary property has 0.7 weight, where the secondary property has 0.3. Using this schema we emphasize the importance of the primary property. Considering these settings our matchmaking method produces the results presented in Table 7.3.

In both weight schema service 16 is successfully identified as the exact match by the matchmaker agent for the service request, since it satisfies both required properties. Additionally the matchmaker agent successfully ranks the services in the second and third categories higher than the services in category four in both schema. The matchmaker agent also determines that the services 1, 2, 3 and 10 does not match to the request, since neither of them satisfy the primary property.

7.3. Comparison of the Two Matchmaking Methods

We can compare the proposed matchmaking methods from the point of view of three different entities in the matchmaking process. These are the service provider, the service consumer and the matchmaker.

The role of the service provider in the matchmaking process is to describe its services accurately and to feed the matchmaker with this information. According to this role of the service provider both of our matchmaking methods are almost equal, since in both methods the service provider should provide the service description through the same structure. In the case of the FSM based method the service provider describes its service through an FSM. In the case of model checking based method, the service provider describes its service through Promela language. Although Promela is more expressive than an FSM in most of the cases it can be easily translated into an FSM. However, in our study we use the process model of OWL-S for the abstraction of this difference. This abstraction is possible, since we can translate OWL-S into an FSM or

Promela model preserving its expressiveness, where OWL-S provides us the required capabilities to describe a service.

The role of the service consumer in the matchmaking process is to describe its required properties from the candidate services and to feed the matchmaker with this information. From the service consumer point of view our matchmaking methods are different than each other. The major difference between the two methods is the representation of the required properties. In the case of FSM based method all the required properties of the service consumer is represented through one common FSM. The main drawback of this method is the increment in the complexity of the FSM, when the number of required properties of the service consumer increases. This increased complexity causes two problems. Firstly, in order to build the FSM, which represents the service consumers requirements, the service consumer should have extensive knowledge about the internal working mechanism of the requested service. In most of the cases the service consumer may not have this knowledge. Secondly, the increasing complexity of the FSM affects the quality and the performance of the matchmaking process. On the other hand, in the case of model checking based method, each specific required property of the service consumer can be represented by an individual temporal logic formula. The advantage of this method is the simplification of the service consumers work while describing its requirements. To describe the required properties the service consumer should only know the parts of the service process that are related to its request.

The matchmaker executes the matchmaking algorithm using the information obtained from the service provider and service consumer. From the matchmaker point of view both methods have its own advantages and disadvantages. The advantage of the FSM based method is its performance. The similarity metrics used by the FSM based method can be implemented efficiently through dynamic programming techniques. One of the most time consuming operation in the FSM based method is the action sequence generation explained in Section 4.2.1. However this process can be performed off-line for the services, which do not affect the performance of the matchmaking operation. Although this method presents good performance, its result quality depends on the

several issues. First of all, the result quality of this method heavily depends on the modeling decisions of the service and the request. As the related case study demonstrates even to change the order of one action significantly affects the matchmaking quality. Another issue is the model detail of the service and request. To get the best result quality, the service and the request should be modeled in the same detail level. However, this causes problems, especially in large domains, where it is hard to obtain a common level of detail considering the service providers and service consumers do not know each other. Another issue for the FSM based method is it dependence on the similarity metrics performance. As the case study results show there is not one best similarity metric for all cases. Each metric may produce good quality results in one case and bad quality results in another.

As the case study results demonstrates the advantage of the model checking based method is its result quality. The main reason of this good quality is the ability of describing the requested properties individually, which makes the service request more accurate by concentrating on the actually requested properties and skipping the unrelated details. In this way the matchmaker knows more precisely what the service consumer really looking for, which increases the quality of the matchmaking results. Additionally, the increased precision of the service descriptions makes the matchmaking process more flexible. This happens, because the service consumer only describes the actually requested properties and leave the decision about unrelated properties to the matchmaker. In FSM based method these unrelated details should also be described by the service consumer, which limits the matchmaker. The main drawbacks of the model checking based method is the alternative property generation process explained in Section 5.2.1. This process is time consuming especially if the depth of the action ontology is large. Another drawback is running the underlying model checking algorithms separately for each requested property on each service.

One common problem for both of the methods is handling of the extra capabilities provided by the service. By extra capabilities we mean the capabilities that are not described in the service request, but given by the service. For instance, when the service consumer looks for a service to buy books, she asks in the service request only

about the capability of the service about selling books. However some book selling services may also have a capability for refund, which is not mentioned in the service request. In such a case it is not clear for the matchmaker whether to match services that have the refund capability in addition to the book selling capability or not. The refund capability may be ignored by the matchmaker, since it does not change the world state, unless it is called by the service consumer. Therefore this extra capability does not have any effect on the matchmaking process. However, in another situation when the service consumer looks for a service to learn the recent air temperature, she creates a service request accordingly. In such a case there may be services, which charges the credit card of the service consumer to provide the air temperature. In this situation it is not clear how to handle these services by the matchmaker. If it matches services, which charges the credit card, the service consumer might be harmed, since she does not expect a credit card charge. On the other hand, if all of the matching services charges the credit card of the service consumer and the matchmaker does not return these services because of the extra capability, the request of the service consumer fails.

8. DISCUSSION

In this thesis we propose two novel service matchmaking methods. In the first matchmaking method we use FSMs to model services and also service requests. We use four structural similarity metrics to determine the services that satisfy the requirements of a service request. In the second matchmaking method we use model checking techniques for matchmaking. In this method we model services in Promela language and service requests through LTL formulae. We use Spin model checker and provide algorithms to use the results of the model checking process for matchmaking purposes. In both methods we use an action ontology to enhance the matchmaking process with semantics, where actions represent atomic operations that we use in the modeling process of services. In the action ontology we map each action to a concept, which provides us information about the hierarchic relations between the actions. We also develop a metric, which we call ACR. ACR provides us a numeric measure, which shows us how much one action can handle functional capabilities of another action. To realize our methods we develop a matchmaking framework. The most important property of this framework is its modular structure, which allows the use of multiple matchmaking methods in conjunction. We performed various case studies to evaluate our matchmaking methods. According to the results of these case studies, we present and discuss the advantages and weak points of our proposed matchmaking methods.

The current service matchmaking architecture on the Internet is based on the UDDI and WSDL standards, where UDDI provides a directory structure for Web service descriptions in WSDL standard according to several criteria. WSDL standard defines how to describe Web services in XML in a machine processable way. However, it does not provide any semantic facilities, which makes human interaction necessary to successfully capture the behavior of the described service. On the other hand UDDI is developed mostly for human beings and does not provide any facilities for machine processing at all, which makes automatic service discovery almost impossible for the current service matchmaking architecture. In the research community there are some attempts to improve WSDL and UDDI, such as importing semantics into WSDL [26]

and improving capabilities of UDDI [27, 28]. However neither of these improvements are widely accepted.

One of the most widely studied matchmaking approaches for automatic service discovery is input-output based matchmaking (alternatively signature or interface matching) [10, 11, 29, 30, 31, 32, 33, 34]. In this approach to find the services that match a service request, the input-output parameters of existing services are compared against the input-output parameters of the service request and the services that have the same parameter set is matched to the service request. To enhance this approach with semantics, input-output parameters are associated with semantic concepts from ontologies and this semantic knowledge is used to find subsumption relations between the request and service parameters. At the end of the matchmaking process the matching services are ranked into different match degrees. Although, different degree approaches are available in the literature, the most common one is the use of coarse grained categories such as exact, plug-in and subsumes. Exact degree shows that the parameters of the service request and the service are identical. Plug-in shows that the parameters of the service request subsumes the parameters of the service. In this case the service can partially handle the request. Subsumes degree shows that the parameters of the service subsume the parameters of the service request. In this case the service provides more general capabilities than the requested capabilities. In OWL-MX [12] Klusch et al. propose an input-output based method through using syntactic similarity metrics from information retrieval in conjunction with the semantic information for better ranking of the matchmaking results. In WSMO-MX Kaufer and Klusch [35] extends this method through adding a filter for matchmaking of pre and post conditions, where they use first order logic to define the pre and post conditions as logic constraints. Şenvar and Bener [36] proposed a layered matchmaking architecture, where they use other information than the service inputs and outputs such as categorization and pre and post conditions and ontological distance information defined and ranked by users. Through this approach their main aim is to directly capture the view of the user for better ranking of results. In general there are two main drawbacks of the input-output based matchmaking approaches.

- *Granularity*: The results of the matchmaking is coarse-grained. That is, the matching services are associated only with some general similarity degrees (exact, plug-in, and so on) and we cannot further discriminate between services that have the same similarity degree. This level of granularity is unacceptable, especially when the number of matching services is large. A better matching approach should provide more precise matching information and should be able to rank the services based on this rating.
- *Precision*: The matchmaking algorithm should provide high precision; meaning that those Web services that are actually labeled as matching should be compatible with the requests. Most input-output matching techniques suffer from low precision, since they do not consider the internal processes of services while performing the matchmaking operation. As a result, different services with identical interfaces are counted as good matches although may they perform completely different tasks.

These two weaknesses of the input-output based matchmaking approach is one of our main motivations to use the internal process models of the services in the matchmaking process.

Dong et al. [37] propose an alternative approach based on clustering techniques. Instead of matchmaking services to requests, main idea of this approach is to find similarities between services. In this way if a service consumer somehow uses a service and finds it useful, she can find similar services through the proposed approach. Authors use inputs, outputs and text descriptions of services to cluster concepts in these information sources and find similar services using these concepts with similarity metrics. However this idea does not provide a complete solution for the matchmaking problem.

Klein and Bernstein [38] propose an indexing mechanism to create a hierarchy of process models, where models are represented using a work-flow language. The process models are defined in an ontology, which is used by both service providers and service consumer. For this purpose the authors rely on the MIT Process Handbook [39]. Authors also develop a query system that works on the hierarchy of process

models for service retrieval purposes. Similar to our approaches, this approach also uses the process models of services instead of the input-output interfaces. However, the indexing and matchmaking mechanisms are totally different than our methods.

Wombacher et al. [40] propose a matchmaking approach, which also uses FSM to model services. For matchmaking they mainly use disjunction, conjunction and intersection relations between FSM models. In this approach, they convert the FSM model of a services into a set of conjunctive FSM models. That is instead of using a single FSM model they use several FSM models, where each model represents a conjunctive branch in the original FSM model. The aim of this approach is to distinguish the implicit disjunction and conjunction relations in the original FSM model and guarantee that all the requested conjunctive branches are satisfied by services. Similar to this approach, in our FSM based method we also separate the original FSM model of the service into several FSM models to represent its branches. However, we do not distinguish between conjunctive and disjunctive relations explicitly in the FSM model level, since we can achieve the same effect through using the ACR metric in the semantic level. Addition to the this, there are two other main differences between our method and this approach. First, it considers only exact matching, that is the services and request that are identical to each other. Hence, there is also no ranking mechanism in this approach. On the other hand, our method supports also partial matches other than the exact matches and provides a ranking mechanism. Second, this approach is based purely on syntactic structures and does not consider any use of semantic concepts. Consequently, the difference between conjunctive and disjunctive relations are embedded into FSM models, which complicates the service models. In our method we separate this knowledge from the FSM model and in this way we simplify the FSM models and the computations on them and make the method more generic through representing this information through semantic concepts.

Best of our knowledge, although model checking methods applied to verification of services [23, 41, 42], they are not used in the matchmaking of services. Hence our model checking based method is unique in the matchmaking of services.

This study can be extended with several improvements. Our FSM based matchmaking method strictly depends on the underlying structural similarity metrics. In our study these heuristics are used individually. An intuitive approach might be using these metrics in conjunction and composing the results to measure the overall structural similarity. According to various cases the effect of each metric can be weighted and these weights can be optimized through learning techniques. The ACR metric has several parameters for fine tuning. These parameters depends also on the underlying ontology structure. Hence the ACR metric should be improved to reduce the effect of these parameters and ontology dependencies. We also leave complete implementation of the proposed framework as a future work, which is an important step to verify this framework. The case studies that we present in this thesis are only representative of the current domains and should be extended for a better evaluation. However, we want to also mention that there is no existing data set that contains the internal process information of services. Hence, we have to design the case studies from scratch. Empirical evaluation of this work will be easier to perform as new process based matchmaking methods are developed.

REFERENCES

1. Singh, M. P. and M. N. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents*, John Wiley & Sons, Chichester, UK, 2005.

2. Mitra, N. and Y. Lafon, *SOAP Version 1.2 Part 0: Primer*, W3C, second edition, April 2007, http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

3. Christensen, E., F. Curbera, G. Meredith, and S. Weerawarana, *Web Services Description Language (WSDL) 1.1*, W3C, March 2001, http://www.w3.org/TR/2001/NOTE-wsdl-20010315/.

4. UDDI Spec Technical Committee, *Universal Description, Discovery and Integration (UDDI) 3.0*, OASIS, July 2002, http://uddi.org/pubs/uddi-v3.00-published-20020719.htm/.

5. Booth, D., H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, *Web Services Architecture*, W3C, February 2004, http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

6. Oaks, P., A. H. M. ter Hofstede, and D. Edmond, *Service-Oriented Computing - ICSOC 2003*, Vol. 2910 of *Lecture Notes in Computer Science*, chapter Capabilities: Describing What Services Can Do, pp. 1–16, Springer-Verlag, Berlin, Heidelberg, 2003.

7. O'Sullivan, J., *Towards a Precise Understanding of Service Properties*, PhD Thesis, Queensland University of Technology, November 2006.

8. Bray, T., J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, *Extensible Markup Language (XML) 1.0*, W3C, fourth edition, August 2006, http://www.w3.org/TR/2006/REC-xml-20060816/.

9. Berners-Lee, T., J. Hendler, and O. Lassila, "The Semantic Web", *Scientific American*, Vol. 284, No. 5, pp. 28–37, 2001.

10. Paolucci, M., T. Kawamura, T. R. Payne, and K. P. Sycara, "Semantic Matching of Web Services Capabilities", *Proceedings of the 1st International Semantic Web Conference*, pp. 333–347, Springer-Verlag, 2002.

11. Li, L. and I. Horrocks, "A Software Framework for Matchmaking Based on Semantic Web Technology", *Proceedings of the 12th International Conference on World Wide Web*, pp. 331–339, ACM Press, New York, NY, 2003.

12. Klusch, M., B. Fries, and K. Sycara, "Automated Semantic Web Service Discovery With OWLS-MX", *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 915–922, ACM Press, New York, NY, 2006.

13. Decker, K., K. Sycara, and M. Williamson, "Middle-agents for the Internet", *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pp. 578–583, 1997.

14. Sipser, M., *Introduction to the Theory of Computation*, Course Technology, second edition, 2005.

15. Huth, M. and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge University Press, 2004.

16. Emerson, E. A., *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, chapter Temporal and Modal Logic, pp. 997–1072, Elsevier and MIT Press, 1990.

17. Guarino, N., *Formal Ontology in Information Systems*, IOS Press, 1998.

18. McGuinness, D. and F. van Harmelen, *OWL Web Ontology Language Overview*, W3C, February 2004, http://www.w3.org/TR/2004/REC-owl-features-20040210/.

19. Martin, D., M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara, *OWL-S: Semantic Markup for Web Services*, W3C, November 2002, http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/.

20. Zaremski, A. M. and J. M. Wing, "Specification matching of software components", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 6, No. 4, pp. 333–369, 1997.

21. Gusfield, D., *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.

22. Holzmann, G. J., *The Spin Model Checker: Primer and Reference Manual*, Addison-Wesley Professional, 2004.

23. Ankolekar, A., M. Paolucci, and K. Sycara, "Spinning the OWL-S Process Model: Towards the verification of the OWL-S Process Models", *Online Proceedings of the ISWC 2004 Workshop on Semantic Web Services: Preparing to Meet the World of Business Application*, 2004, http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-119/.

24. Patel-Schneider, P., *A Proposal for a SWRL Extension towards First-Order Logic*, W3C, April 2005, http://www.w3.org/Submission/2005/SUBM-SWRL-FOL-20050411/.

25. Horrocks, I., P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*, W3C, May 2004, http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/.

26. Akkiraju, R., J. Farrell, J. Miller, M. Nagarajan, M. Schmidt, A. Sheth, and K. Verma, *Web Service Semantics-WSDL-S*, W3C, November 2005, http://www.w3.org/Submission/2005/SUBM-WSDL-S-20051107/.

27. Paolucci, M., T. Kawamura, T. Payne, and K. Sycara, "Importing the Semantic Web in UDDI", *Proceedings of the Web Services, E-Business and Semantic Web Workshop, CAiSE 2002.*, pp. 225–236, Springer-Verlag, 2002.

28. Srinivasan, N., M. Paolucci, and K. Sycara, "Adding OWL-S to UDDI, Implementation and Throughput", *Proceedings of the 1st International Workshop on Semantic Web Services and Web Process Composition*, 2004.

29. Benatallah, B., M. S. Hacid, A. Leger, C. Rey, and F. Toumani, "On Automating Web Services Discovery", *International Journal on Very Large Data Bases*, Vol. 14, No. 1, pp. 84–96, 2005.

30. Keller, U., R. Lara, H. Lausen, A. Polleres, and D. Fensel, "Automatic Location of Services", *Proceedings of the 2nd European Semantic Web Conference*, pp. 38–49, Springer-Verlag, 2005.

31. Kifer, M., R. Lara, A. Polleres, C. Zhao, U. Keller, H. Lausen, and D. Fensel, "A Logical Framework for Web Service Discovery", *Online Proceedings of the ISWC 2004 Workshop on Semantic Web Services: Preparing to Meet the World of Business Application*, 2004, http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-119/.

32. Sycara, K., S. Widoff, M. Klusch, and J. Lu, "Larks: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace", *Autonomous Agents and Multi-Agent Systems*, Vol. 5, No. 2, pp. 173–203, 2002.

33. Bansal, S. and J. M. Vidal, "Matchmaking of Web Services Based on the DAML-S Service Model", *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 926–927, ACM Press, 2003.

34. Colucci, S., T. D. Noia, E. D. Sciascio, M. Mongiello, and F. M. Donini, "Concept Abduction and Contraction for Semantic-based Discovery of Matches and Negotiation Spaces in an E-marketplace", *Proceedings of the 6th International Conference*

on Electronic Commerce, pp. 41–50, ACM Press, New York, NY, USA, 2004.

35. Kaufer, F. and M. Klusch, "WSMO-MX: A Logic Programming based Hybrid Service Matchmaker", *Proceedings of the 4th IEEE European Conference on Web Services*, pp. 161–170, 2006.

36. Şenvar, M. and A. Bener, *Advances in Information Systems*, Vol. 4243 of *Lecture Notes in Computer Science*, chapter Matchmaking of Semantic Web Services Using Semantic Distance Information, pp. 177–186, Springer-Verlag, 2006.

37. Dong, X., A. Y. Halevy, J. Madhavan, E. Nemes, and J. Zhang, "Simlarity Search for Web Services", *Proceedings of the 13th International Conference on Very Large Data Bases*, pp. 372–383, Morgan Kaufmann, 2004.

38. Klein, M. and A. Bernstein, "Toward High-Precision Service Retrieval", *IEEE Internet Computing*, Vol. 8, No. 1, pp. 30–36, 2004.

39. Malone, T. W., G. A. Herman, and K. Crowston, *Organizing Business Knowledge: The MIT Process Handbook*, Mit Press, 2003.

40. Wombacher, A., P. Fankhauser, B. Mahleko, and E. Neuhold, "Matchmaking for Business Processes Based on Conjunctive Finite State Automata", *International Journal of Business Process Integration and Management*, Vol. 1, No. 1, pp. 3–11, 2005.

41. Fu, X., T. Bultan, and J. Su, "Formal Verification of e-Services and Workflows", *Proceedings of the Web Services, E-Business and Semantic Web Workshop, CAiSE 2002.*, pp. 188–202, Springer-Verlag, 2002.

42. Singh, M. P., "Distributed Enactment of Multiagent Workflows: Temporal Logic for Web Service Composition", *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 907–914, 2003.