ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA
CAMPUS DI CESENA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

LOGIC-BASED COORDINATION: A SEMANTIC APPROACH TO SELF-COMPOSITION OF SERVICES

Relatore:
Prof. Andrea Omicini

Correlatore:
Prof.ssa Giovanna Di Marzo Serugendo

Dott. Giovanni Ciatto

Presentata da:
Ashley Caselli

Sessione III
Anno Accademico 2017/2018

Keywords

Multi-agent systems
Logic programming
Logic-based coordination
Self-composition of services

To my beloved family

Contents

List of Figures

Listings

Abstract

Logic-based approaches have always been at the core of research concerning the coordination of multi-agent systems (MAS). Starting from the Shared Prolog, logic-based coordination models have evolved into comprehensive approaches for nowadays complex and distributed systems, such as IoT (e.g. ReSpecT) and self-organizing ones (e.g. Logic Fragment Coordination Model). Separately and in parallel to the emergence of MAS, research in the Web field has been focusing on providing technologies in support of the creation of Internet-based distributed systems in which automatic processes such as service discovery, invocation and composition are feasible. Integrating MAS and Web paradigms will help enable new and advanced operational and usage modalities of Web services, and vice versa. Those operational modalities, such as self-adaptation and self-management, are fundamental in today's scenarios characterized by dynamism. It is therefore presented a logic-based coordination model in which the self-composition of semantically annotated services is highly promoted and supported. A possible implementation is also provided in the form of a basic prototype developed using tuProlog, `TuCSoN` and `ReSpecT`$\mathbb{X}$. Moreover, the assessment of the model is illustrated through formally defined scenarios.

Introduction

Agent technology is a software paradigm that permits to implement large and complex distributed applications [30], worldwide referred as multi-agent systems (MAS). Agents, namely encapsulated computer systems situated in some environment and capable of flexible and autonomous actions in that environment [1], require interaction to meet their stated objectives. Research have mostly focused on the coordination mechanisms, specifically how to rule the environment in which they are situated. Logic-based approaches have always been at the core of those research concerning the coordination of multi-agent systems (MAS). Starting from the Shared Prolog, logic-based coordination models have evolved into comprehensive approaches for nowadays complex and heterogeneous distributed systems.

Separately and in parallel to the emergence of MAS, the Web field has been focusing on providing technologies in support of the creation of Internet-based distributed systems. Service-oriented programming has dramatically changed the way software applications are developed [16]. Nowadays, indeed, service-oriented architectures (SOA) represent the standard approach for distributed systems engineering [7].

At a glance, both MAS and SOA paradigm support the idea of distributed autonomous entities [49]. However, unlike Web services which provide functionality through simple executable methods, agents that act intelligently use knowledge to react to and act on their environment autonomously and proactively [32]. On one hand, Web service technologies enable automatic processes such as service discovery, invocation and composition. On the other hand, agents provide a unique capability in mediating user goals to determine service invocations [32]. Integrating Web services and software agents might result in a fruitful match, taking advantages of their functionalities one from the other. Furthermore, once this interconnection is established, software agent concepts and technologies will help enable new and advanced operational and usage modalities of Web services, and vice versa. [26].

With the dynamism that characterizes today's systems, the user's need have to be met by exploiting available resources, even when an exact match does not exist [31]. The run-time evolution of resource discovery must be supported and self-autonomous behaviors must be exploited to meet the users requests. For instance, it is the case of the research in service composition approaches that have led to methods and technologies to obtain self-composable services.

"A challenging problem is to compose services dynamically, on demand."[3]

An effective approach to address the quoted challenge turns out to be the combination of the technologies provided by SOA and MAS fields. As a matter of fact, in Web services environments with semantically annotated services, software agents are important entities that facilitate user's tasks in a transparent manner. [32] To this aim, self-autonomous composition processes might be obtained by borrowing tools from both fields. In particular, leveraging on tools wherewith a service may be semantically annotated and the autonomous reasoning capabilities each agent owns, in order to compute semantic-based reasoning on which future decisions will rely.

This dissertation contributes with designing a novel coordination model that promotes and supports spontaneous service composition within a multi-agent systems (MAS) environment. The model is grounded on logic-based computations, exploited to compute semantic relations that lead the matchmaking among services.

The remainder of this dissertation is organized as follows. Chapter 1 provides an overview of the logic-based coordination and self-composition current approaches. Chapter 2 presents the designed coordination model. It is conveyed using a formal approach (i.e. process algebra) in order to tackle the major problem of the typical approaches used to define service compositions, namely the verification of processes' correctness [52]. Chapter 3 contains a basic prototype which aims to provide a possible implementation of the model. It only includes the core functionalities that are implemented by using `TuCSoN`, tuProlog and the `ReSpecTX` language. Finally, in Chapter 4 the assessment of the model is argued through both generic and specific scenarios. For each scenario the operations involved are argued by showing the computational steps that lead to the shown solution, using a formal representation.

Chapter 1

State of the art

This chapter summarizes the key points concerning coordination models and the approaches adopted to tackle the dynamic service composition challenge. Regarding the coordination models, the focus is made on the logic-based approaches that, starting from the Shared Prolog, have evolved into comprehensive ones for complex systems (such as IoT, e.g. `ReSpecT`) and self-organizing systems (e.g. Logic Fragment Coordination Model - LFCM [14]). On the other hand, due to the dynamism and openness that characterize modern scenarios, service composition approaches have gone towards autonomous computations that give properties such as self-adaptation and self-management to the system.

1.1 Coordination models

"Coordination models have proven useful for designing and implementing distributed systems." [19]. Distributed system in which heterogeneous entities coexist is a reality of our days. These entities must seamlessly integrate and coordinate among each others to achieve the desired and expected goals. To this purpose, starting from the interaction on shared variables, through message passing, more recent approaches consisting of using higher-level programming models and languages were coined; those are called coordination models and languages. During the years several definitions have been provided:

"A coordination model is the glue that binds separate activities into an ensemble."[13]

"A coordination model provides a framework in which the interaction of active and independent entities called agents can be expressed." [10]

to name a few.
The purpose is providing a means of integrating a number of components together, "by interfacing with each component in such a way that the collective set forms a single application that can execute on and take advantage of parallel and distributed systems" [48]. To this purpose, according to [10], the issues to tackle are represented by managing the creation and destruction of agents, communication among agents, and spatial distribution of agents, as well as synchronization and distribution of their actions over time.

Identifying components of which the coordination models are built might be a constructive approach to define them. A coordination model can be thought as made of three elements [11]:

- **Coordinables** Entities whose interaction is ruled by the model, namely the coordinated entities

- **Coordination media** Abstraction in which the coordination is made possible. Examples might be simple media as semaphores, channels, monitors, or more complex like blackboards, tuple spaces, etc. Moreover, it can be used to aggregate agents and manipulate them as a whole

- **Coordination laws** A coordination model should define a finite set of rules to describe how the coordination of the agents occurs through the media. These are defined in terms of a *communication* and a *coordination* language that define respectively the syntax of the exchanged data structures and the primitive used to interact (with their semantics)

Basically, a coordination model provides a framework where the interaction of entities can be expressed. Entities are called agents and are active and independent.

Coordination models can be classified using several dimensions. A useful classification consists in discerning between the two major categories of coordination programming, namely either *data-driven* or *control-driven* (also called process- or task-oriented) [48]. Briefly, within data-driven coordination model the entities are in charge of both examining and manipulating data as well as coordinating themselves and/or other entities by invoking the coordination mechanism provided by the language. In fact, purely data-driven coordination models do not clearly distinguish between coordination and computational processes. The coordination media's content might result in a combination of data and code. It is thus up to the programmer to design their code in such a way that the two concerns are clearly separated, even though the model does not provide this separation at the syntactical level. On the other hand, control-driven coordination models present an almost complete separation between computational and coordination processes. Usually, the latter is achieved by exploiting an ad-hoc coordination language and coordinated entities treated as black-boxes that provide a well-defined input/output interface.

1.1.1 Logic-based coordination

Logic based approaches have always been at the core of research on coordination, especially regarding multi-agent systems (MAS). When it comes to formalize the coordination mechanisms and primitives, logic can play a key role. For example, first-order logic and unification of unitary clauses and ground terms represent a powerful mechanism of matching exploited by tuple-based coordination models. In addition, different approaches such as temporal logic and spatial logic can express further aspects of coordination, especially in the verification of emergent global properties [15].

Known logic-based coordination models are PoliS [9], $\mathcal{ACLT}$ [17] and `TuCSoN` [46, 43].

PoliS extends the Linda approach [22] to parallel programming. It is based on multi-set rewriting in which coordination rules produce and consume multi-sets of tuples.

Within the $\mathcal{ACLT}$ model (Agents Communicating through Logic Theories), the tuple space is seen as container of logic theories. The access is granted to logic agents in order to perform deduction processes. The matching mechanism used is first-order logic and unification of unitary clauses and ground atoms.

`TuCSoN` borrows $\mathcal{ACLT}$ concepts and defines a model for distributed processes and agents, introducing the idea of tuple center as an engineered tuple space whose behavior can be modeled using a reaction specification language.

1.1.2 `TuCSoN`

Taking inspiration from $\mathcal{ACLT}$, `TuCSoN` (Tuple Centers Spread over the Network) defines a model for distributed processes, as well as autonomous, intelligent and mobile agents. Each node is associated to a tuple center that is "a tuple space whose behavior can be defined by means of reactions to communication events" [45]. In fact, tuple centers are tuple spaces enhanced with the notion of behavior specification, thanks to the `ReSpecT` language [45]. The latter is a reaction specification language, based on first-order logic tuples. Such a language is used to link computational tasks with the basic Linda-inspired coordination primitives: `in`, `out`, `rd`, `inp` and `rdp` [45]. An important property of `TuCSoN` is the openness regarding to the programming languages it supports. In fact, like other pure coordination models, `TuCSoN` is not associated to any specific programming language, thereby it supports agents written in several languages (e.g. Java, C).

1.2 Web services

"Web services are a consolidated reality of the modern Web with tremendous, increasing impact on everyday computing tasks" [35]. According to this statement, it is reasonable to think that the traditional Web has been strongly affected and changed with the coming of Web services. Web service, in fact, have deeply changed the Web, enhancing it by providing a new level of functionality and turning it into the largest and most accepted distributed computing platform ever.

“Nowadays, Web can be viewed not only as a distributed source of information, but also as a distributed source of services” [36]. Nevertheless, their full power of integrating into applications or composite services has not been thoroughly used yet.

According to [6], a Web service is a software system designed to support interoperable machine-to-machine interaction over a network and it has an interface described in a machine-processable format. Beyond that, several definitions have been provided during the years that can be summed up into the following points. Any service that

- is available over the network
- uses a standardized messaging system
- is not tied to any programming language or operating system
- is self-describing
- is discoverable

might be defined as a Web service.
Basically, they enable communication among various applications by using open standards, e.g. HTML, XML, SOAP, WSDL (SOA approach), URI and HTTP (REST approach). Both of approaches allow to represent Web services in a standardized fashion, in terms of their interfaces. For instance, considering the SOA approach, WSDL is merely a standard language used to describe the means of interacting with offered services. Likewise, by adopting a REST approach, it is possible to represent a Web service as a resource whose URI is well-known; thus once again it is being described the means of interacting with it rather than other aspects, such as the service capabilities. All the standards used to define a service work at a syntactic level. In fact, they lack of expressiveness when it comes to define service’s capabilities and requirements in an unambiguous and machine-readable manner.

1.2.1 Syntactic description limitation

Considering WSDL, it only allows to describe the functional and syntactic aspects of a service. It is used to describe a Web service by the parameters, associated with abstract data types, and operations it supports. However, the usage of this description brings limitations. For instance, assuming the space in which services are published is populated by agents, namely any piece of software that works autonomously and proactively [2]. Their autonomous behavioral aspect is strongly limited on services, since they can only see the parameters names and abstract data type and cannot deduct what the service actually provides, as a human could.
Moreover, assuming the existence of two services that provide the same functionalities with descriptions that syntactically differ. In this case, since any agent can only reason about the syntactic definition provided, the two services will never be considered as equivalent.

The description of a service by means of its syntactic parameters severely limits the execution of automated tasks on services, such as discovery, invocation and composition. To this purpose a machine-readable representation of the service is required.

1.2.2 Semantic Web services

When introducing the Semantic Web Tim Berners Lee stated "The introduction of ontologies can enhance the functioning of the web in many ways" [5]. The birth of the semantic Web doubtlessly changed the way the Web can be used, especially by such entities that are capable of autonomous computations. The semantic Web transformed the Web into a repository of computer readable data [8], as a matter of fact. As years went by, the integration among Web services and semantic Web tools had been increasing, leading to the birth of the so-called Semantic Web services. The approach consists of enriching the traditional Web services with rich formal descriptions [28]. The latter copes with the lack of expressiveness the traditional description languages have. Usage of the semantic Web tools to represent a Web service surely opens to the possibility of performing automated computations on

many aspects. Specifically, location, composition and mediation can become a dynamic process.
For instance, to locate the best services that are able to solve a particular problem or to automatically compose the relevant services to build applications dynamically [20].

The advent of Semantic Web services allows agents to perform automated reasoning that is grounded on a semantic matching computation between the published service description and the request. Following the same approach could then be possible perform tasks such as the location and composition of services on the fly. Three main approaches have been driving the development of Semantic Web services frameworks are: IRS-II [42], WSMF [21] and OWL-S [37, 39, 36].

1.3 Composition of services

Service composition is known as the mechanism of combining two or more basic services into a possibly complex service [31]. Coupling together Web services and semantic Web technologies might help to step forward toward an improvement of the Web services composition exploiting rich and machine-readable representation of service properties and capabilities. In addition, always exploiting those technologies, it might be possible to perform a reasoning mechanism in order to select and aggregate services [40].

Semantic Web services have brought the chance to redesign Web services, creating a more expressive representation under many standpoints. Main approaches previously cited allow to express the service in terms of its functional and non-functional parameters, giving the opportunity to perform automated reasoning. Here, the focus is made on the automated composition process, which is defined as the property of the system to locate and compose services on the fly. The key aspects of the process reside in how the services are selected and bonded. Hence, the composition process should be seen as an outcome of two minor phases: *selection* and *binding*.

Selection is the phase where a concrete service is sought and identified to be used in a composition. It may occur in three different stages of the composition lifecycle: *design time*, *deployment time*, and *run time* [35].

Binding is the phase where the services are actually connected, so as to create the composite and more evolved ones. If it occurs at design time the composition is called *static composition*; in both the other stages it is called *dynamic composition* [35].

1.3.1 Static composition

Design time

The composition is built during the design time of the system by the hand of the developer. They choose the services once for all and the selected services are permanently connected unless they decide to modify the actual configuration. Obviously, this kind of composition leads to correct compositions but completely lacks of scalability since it requires human intervention to adapt the system to changes.

1.3.2 Dynamic composition

Deployment time

During a deployment time composition process the binding occurs whenever a service shows up, claiming to be ready to provide its services. This type of composition requires the system to compute all the possible compositions among the available services and redesign them every time new services enter the system.

Run time

A run time composition process doubtlessly requires less computational resources since it only computes the composition when a request reaches the system. It also improves the scalability of the system, coping with the problem of being prone to changes. However although provides a better scalability, it surely adds overhead in the request's computation, that might bother the final user.

1.4 Self-composition of services

Self-composition of services is a completely decentralized process, hence favoring scalability, whereby services are able to compose with other services autonomously. The process aims to create higher level functionalities within the system by leveraging the available resources. It is highly scalable and copes with dynamic appearance and disappearance of components within the system [16]. Composition might be carried out either using a syntactic approach, as in [16], or leveraging the tools and technologies provided by semantic Web, as previously mentioned. The latter approach indeed overcomes the syntactic one since it performs a reasoning process that takes into account formal and more expressive data, such as ontologies. By means of the ontologies, in fact, it is possible to represent both functional and non-functional aspects, as well as behavioral aspects that more accurately describe a service. Automated reasoning is grounded on semantic matching among formal concepts. To this purpose the matching policies play a key role in the process and should be accurately designed.

Benefits brought in are automatic discovery, invocation, composition and monitoring. On the other hand, the need of new computational processes that support and implement those capabilities arises, as well as new problems to tackle. Further development of frameworks (such as [53] and technologies might mitigate and solve the open problems, and presumably lead to more and more reliable automatic processes.

Chapter 2

Formal model

Within the chapter a definition of the coordination model is argued. As the title suggests, the model is conveyed using a formal approach (i.e. process algebra) due to the benefits it brings.

The coordination model is composed of two main entities: a blackboard and external entities, namely “agents”. The blackboard is considered as a space exploited by the agents as coordination medium. Therefore, an agent interacts with other agents in the system by means of the blackboard. Agents are categorized in two basic groups according to their capabilities:

- A **service agent** provides services via the system
- A **user agent** exploits the services available in the system

Whenever an agent is willing to communicate with others, it leaves or consumes a message on the blackboard. According to the category an agent belongs, there exist predefined messages that it may leave or consume. For instance, a user agent might want to prove if there is any service that is currently able to fulfill a request. A request message is left on the blackboard and consumed by the service able to fulfill it. However, the latter statement is not completely correct. Since within the model the self-composition of services based on semantic is promoted, the request message might be fulfilled either by a simple service or a composed one.

Note that it is implicitly assumed that only a single global blackboard exists in the system, while in a typical modern scenario it is likely to have a network of blackboards. For simplicity, the model describes a node as a component that only consists of one blackboard.

2.1 Syntax

Within the section the syntax of the model is provided in the form of *extended Backus-Naur form* (EBNF) notation. The latter notation is one of the notations that can be used to express a *context-free grammar* (CFG) which is mostly used to make a formal description of a formal language (e.g. a computer programming language).

Definition 2.1.1. An EBNF grammar consists of

- a finite set T of terminals
- a finite set N of non-terminals disjoint from T
- a start symbol $S \in N$
- a finite set of rules $A ::= \tau_1 \mid \ldots \mid \tau_t$, at most one for each non-terminal $A \in N$, where each $\tau_k, 1 \leq k \leq t$, is a base, bracketed or concatenated regular expression over the set $T \cup N$

2.1.1 Service descriptors

A *service agent* is defined as the entity able to provide one or more services via the blackboard. In order to better explain the defined syntax, just one service agent will be considered.
Each service provided by a service agent is described using a *service descriptor*. It is a notation that includes the functional parameters of the described service, that are:

- the set of the *inputs (I)* it accepts
- the *output (O)* it provides

A service is therefore modeled as a black-box element that accepts one or more inputs and always provides one output.

$$\begin{aligned}
SD &::= \texttt{service}(Q) \mid SD \overset{N}{\texttt{argof}}\ SD && \textit{service descriptor}\\
I &::= \epsilon \mid N : T \mid I, I && \textit{input}\\
O &::= \epsilon \mid T && \textit{output}\\
N &::= n_1 \mid n_2 \mid n_3 \mid \dots && \textit{name}\\
T &::= t_1 \mid t_2 \mid t_3 \mid \dots && \textit{type}
\end{aligned}$$

Term *SD* is the service descriptor, which is expressed using Q in case it describes a simple service. As it will be defined in the following pages, the Q term is nothing less than a composition of inputs I and output O.
Term I expresses the input parameters using a notation *name:type*, where the *name* is given by any production of N and the *type* by any production of T. Term N defines any plain string, whilst T represents the datatype. The latter is meant to be mapped into a concept of an already existing ontology, in order to be used in a semantic reasoning process.
Note that both terms I might be a sequence by means of the syntactic operator ",". Regarding the second production of the *SD* term, it is used to represent a composition made of two (or more) service descriptors. To this purpose, the `argof` operator plays as syntactic operator that connects two services. Likewise, `service` operator is used for the mere sake of facilitating the reading of the grammar.

2.1.2 Services

A service agent expresses its willingness to be involved in an interaction publishing its services onto the blackboard. Therefore, an unpublished service cannot be involved in any interaction within the system because it cannot serve any possible request. The basic operations a service agent has at its disposal are the *publish* and the *serve*, syntactically defined as follow:

$$S ::= \texttt{publish}(SD) \mid \texttt{serve}(\texttt{service}(Q)) \mid S \cdot S \qquad \textit{service}$$

Simply, a service is able to perform two operations: publish and serve. The $\texttt{publish}(SD)$ production expresses the operation during which the service agent leaves on the blackboard the service descriptor *SD* of each service it holds, in order to be discovered as active services within the system. While the $\texttt{serve}(\texttt{service}(SD))$ term represents the phase in which the specific service descriptor *SD* is served (keep in mind that a service descriptor $SD = \texttt{service}(Q)$).
Finally, the production $S \cdot S$ is defined to express the case in which a service agent performs two or more events consequently.
The `publish` and `serve` operators are mere syntactic sugar of the grammar. The above grammar is subjects to the following axioms:

$$S \cdot S' \not\equiv S' \cdot S \tag{2.1}$$

$$S \cdot (S' \cdot S'') \equiv (S \cdot S') \cdot S'' \tag{2.2}$$

Basically, the "·" operator is defined as not commutative and associative.

2.1.3 Blackboard

The blackboard is defined as the space used as coordination medium by the agents. Each agent can perform basic read/write actions on the blackboard, thus leave or consume messages. It follows a righteous representation of the blackboard might be as a set whose elements are the messages.
The following grammar describes its syntax:

$$B ::= \emptyset \mid SD \mid \texttt{call}(C) \mid \texttt{in\_call}(C, SD) \mid B \cup B \qquad \textit{blackboard}$$

Blackboard is defined as a set containing a service descriptor *SD*, a $\texttt{call}(C)$ or another blackboard *B*. Assuming that there exists only one blackboard *B* within the system, then the latter assertion should be interpreted as if a blackboard is a set of messages posted on it over the time. The blackboard could also be in the empty state which is denoted by the $\emptyset$ terminal symbol.

The "$\cup$" operator is subject the following properties:

$$\begin{aligned} B \cup B' &\equiv B' \cup B \\ B \cup (B' \cup B'') &\equiv (B \cup B') \cup B'' \\ B \cup \emptyset &\equiv B \end{aligned}$$

Briefly, it is defined as a commutative and associative operator, with $\emptyset$ as the identity value.

2.1.4 Requests

A *request* is a message left on the blackboard by an agent. The model presents two different syntax to differentiate the requests: *query* and *call*. As the names already suggest, they are slightly different and their difference has to be searched into the purpose of each. A *call* represents an actual call that will be left on the blackboard to be managed by one service or a composition of them. Instead, a *query* represents an exploration that is a request used to prove, at the time it is made, whether the system owns or not the capabilities needed to fulfill it.
The existence of a *request* implies that eventually there will be a *response*. According to the request's type, the *response* could be either a plain or a

boolean value.

$$\begin{array}{rll}
\mathit{Req} ::= & \texttt{query}(Q) \mid \texttt{call}(C) & \textit{request} \\
Q ::= & I,\ O & \textit{query} \\
C ::= & A,\ O & \textit{call} \\
A ::= & \epsilon \mid N : T(V) \mid A, A & \textit{arguments} \\
V ::= & v_1, v_2 \dots v_n & \textit{terminal values} \\
\mathit{Res} ::= & \texttt{res}(\mathit{Const}) \mid \texttt{res}(V) & \textit{response} \\
\mathit{Const} ::= & \top \mid \bot & \textit{boolean value}
\end{array}$$

Term *Req* and *Res* represent respectively the *request* and the *response*. As mentioned, a request could be a *query* or a *call*. In fact, term Q expresses an exploratory request (query) and it is defined as the composition of inputs I and output O; where I and O are exactly the same as defined in subsection 2.1.1. Instead, term C represents a call request and it is expressed as a composition of arguments A and output O. The difference in the representation among a query and a call foretells the fact that a query request just provides datatype to the system; thus will work in an higher and abstract level. On the other hand, a call request piggybacks the actual values with which the services able to fulfill it will be fed in order to provide a response.

Finally, *Res* is the *response* and it is represented using a boolean constant *Const* or a value V. The boolean constant represents the boolean truth or falsity, which is the response that will be given to a query request. On the contrary, the response given to a call request is described by the value V. Note that `query`, `call`, and `res` are syntactic sugar.

2.1.5 Users

A user agent expresses its willingness to interact with other agents leaving or consuming messages concerning a request from the blackboard. The user agents are meant to be the request performer agents, therefore the syntax allows to express the handling of the request and response. The operations that a user agent performs simply are: leaving request messages on the blackboard and consuming response messages from it.

$$U ::= \mathit{Req} \mid \mathit{Res} \mid U \cdot U \mid \texttt{halt} \qquad \textit{user}$$

The production *Req* expresses the operation with which the user leaves a request message on the blackboard; while the *Res* expresses the one where the user consumes the response message. Considering that a user agent could perform two or more events consequently, as defined for the service agents, the production U · U is used to describe the aforesaid situation.
At last, the production `halt` is used to handle the eventual termination event. The "·" operator has already been defined in axioms 2.1 and 2.2.

2.1.6 System

As already mentioned, the system is composed by two entities, namely the *blackboard* and the *agents*. Moreover, the agents are in turn categorized into *service agents* and *user agents*. Therefore, the whole system may be described as the composition of three elements that coexist in time: *blackboard*, *service agents* and *user agents*.
The following grammar shows the syntax defined for the system:

$$\begin{aligned} \mathit{Sys} &::= B \parallel U_S \parallel S_S && \textit{main system} \\ S_S &::= S \parallel S_S && \textit{list of services} \\ U_S &::= U \mid (U \parallel U_S) && \textit{list of users} \end{aligned}$$

The whole system, expressed by means of the *Sys* term, is an aggregation of three parallel entities: a blackboard B, a list of users U_S and a list of services S_S. These, in turn, are respectively an aggregation of user and service agents. Conversely, the blackboard B is defined unique within the system.
The grammar presented above is subject to the following axioms:

$$S \parallel S_S \not\equiv S_S \parallel S$$
$$B \parallel (U_S \parallel S_S) \equiv (B \parallel U_S) \parallel S_S$$

Basically, the "$\|$" operator is defined as not commutative and associative.

2.1.7 Labels

The following grammar describes the set of labels that can be used in the LTS (labeled transition system) exploited to describe the operational semantics (see section 2.2). The definition of the label's syntax gives the freedom to easily extend the transition rules of the LTS, just adding new labels to the provided grammar.

$$\begin{aligned} E ::= {} & \textit{publish\_sd} \mid \textit{publish\_call} \mid \textit{serve\_call} \mid \textit{in\_call} \mid \\ & \textit{serve\_in\_call} \mid \textit{last\_in\_call} \mid \textit{prove} \mid \textit{compose} \mid \tau \end{aligned} \tag{2.3}$$

In order to ease their comprehension, the labels are mostly defined using the same name of the transition rule they are involved in. Only exception is made for τ, used in the silent transition defined as `[DECAY]` (see section 2.2.1.

2.2 Operational semantics

The operational semantics is given as a LTS (Labeled Transition System). It is a tuple $(S, \rightarrow, \Lambda, s_0)$, where

- S is a set of states
- $\rightarrow$ is a transition relation, $\rightarrow \subseteq (S \times \Lambda \times S)$
- Λ is a set of labels
- s_0 is the initial state, $s_0 \subseteq S$

To increase the transitions readability the following notation will be used from now on:

$$\rightarrow (s_1, \lambda, s_2) \Leftrightarrow s_1 \xrightarrow{\lambda} s_2$$

where

$$
\begin{aligned}
s_1, s_2 &\in S \\
\lambda &\in \Lambda
\end{aligned}
$$

The latter notation expresses a transition from state s_1 to state s_2 with label λ.
According to the previous definition of the model's syntax, the LTS is subject to the following properties:

$$
\begin{aligned}
S &\equiv \mathcal{L}(Sys) \\
\Lambda &\equiv \{\tau\} \cup \mathcal{L}(E)
\end{aligned}
$$

with *Sys* defined in subsection 2.1.6, E defined in 2.3 and $\{\tau\}$ defined as the silent transition (see more in subsection 2.2.1).

Definition 2.2.1. Let $\mathcal{L}$ be the set of all strings that can be derived from a grammar G. It is called the language generated by G and formally defined as follows:

$$\mathcal{L}(G) = \{W \mid W \in \Sigma^*,\ S \overset{G}{\rightarrow} W\},$$

where Σ^* is the set of all strings that belong to the alphabet Σ.

2.2.1 Rules

Transition relations simply model the effect of executing an action on the blackboard. As largely discussed the blackboard is used as coordination medium by the agents, that actively leave or consume messages to interact among them. The distinction between a service and a user agent also denotes the capabilities they have in terms of interaction. It comes naturally that sometimes the messages left by one, rather than the other, might be correlated in time because one cannot happen without another or vice versa. For instance, a user's request cannot be handled if a service that is able to

fulfill it has not been published on the blackboard yet.
Briefly, the legal actions are

- publication of a service descriptor SD on the blackboard B
- publication of a call request $\texttt{call}(C)$ on the blackboard B
- serve of a call request $\texttt{call}(C)$
- proof of an exploration request $\texttt{query}(Q)$
- composition of two or more services
- decay of service descriptor SD within the blackboard

In the following paragraphs, they will be shown and explained more in detail.

Service descriptor publication

Operation [PUBLISH-SD] plays the role of publishing a service descriptor SD on the blackboard.

$$\texttt{publish}(SD) \cdot S \parallel S_S \parallel U_S \parallel B \xrightarrow{publish\_sd} S \parallel S_S \parallel U_S \parallel B \cup SD \quad \texttt{[PUBLISH-SD]}$$

It occurs without preconditions, hence it may be triggered anytime during the system lifetime. It is performed by a service agent that is willing to publish the offered services within the system. The execution of the operation affects the blackboard state, enriching it with the service descriptor SD.

Call request publication

Operation [PUBLISH-CALL] denotes the action made by a user agent of publishing a call request on the blackboard.

$$\texttt{call}(C) \cdot U \parallel S_S \parallel U_S \parallel B \xrightarrow{publish\_call} U \parallel S_S \parallel U_S \parallel B \cup \texttt{call}(C) \quad \texttt{[PUBLISH-CALL]}$$

Likewise the publication of a service, it may occur without any precondition. Execution of the operation affects the blackboard state, enriching it with a new call request $\texttt{call}(C)$.

Serve

Within this section the operations concerning the serve of a call request are given. In more details, the serve operation is split up into four operations: [SERVE-CALL], [IN-CALL], [SERVE-IN-CALL] and [LAST-IN-CALL].

Operation [SERVE-CALL] is the operation that serves a call request that can be served by a single service. Whilst, in case of a composed service CS exists and it is the one capable of performing the computation, the execution is split up into $n+1$ steps (with $n =$ number of services that compose CS). The n steps are the ones required to execute each single service involved in the composition; whilst the additional one is a sort of syntax modeling step in order to support the services computational chain. Operation [IN-CALL] starts the chain of executions. It is followed by a sequence of [SERVE-IN-CALL] operations which end by the [LAST-IN-CALL] operation that provides the result to the user.

[SERVE-CALL] operation is an atomic operation that is triggered each time the call request may be fulfilled by a single service, i.e. not composed. The operation executes the selected service and provides the result to the user.

$$\frac{SD = \mathtt{service}(I,O) \wedge \mathit{typeof}(\mathtt{call}(C)) \sim SD \wedge V = \mathit{execute}(\mathtt{serve}(SD), \mathtt{call}(C))}{\mathtt{serve}(SD) \cdot S \parallel S_S \parallel U_S \parallel B \cup SD \cup \mathtt{call}(C) \xrightarrow{\mathit{serve\_call}} S \parallel \mathtt{res}(V) \cdot U \parallel S_S \parallel U_S \parallel B \cup SD} \quad \texttt{[SERVE-CALL]}$$

[IN-CALL] operation is in charge of redesigning the call request in order to support the execution of the services involved in the composition. The operation is indeed executed each time the sent call request can be fulfilled by a composed service. During its execution, the blackboard state is modified and enriched with a new call message in_call that contains the service descriptor SD of the first service to be executed in the composition, in addition to the original call request $\mathtt{call}(C)$.

$$\frac{SD = SD' \overset{N}{\mathtt{argof}} SD'' \wedge \mathit{typeof}(\mathtt{call}(C)) \sim SD}{S \parallel S_S \parallel U_S \parallel B \cup SD \cup \mathtt{call}(C) \xrightarrow{\mathit{in\_call}} S \parallel U \parallel S_S \parallel U_S \parallel B \cup SD \cup \mathtt{in\_call}(C, SD')} \quad \texttt{[IN-CALL]}$$

[SERVE-IN-CALL] operation is in charge of partially fulfilling the call, executing the selected single service of the composition. However, the execution result is not provided to the user yet, but it is added as input parameter to a new `in_call` message that is afterwards published on the blackboard. The new message is hence containing the service descriptor of the following service to be executed in the composition and a call that is composed of:

- the computed result V in the form of an argument generated by the A production; hence $N : T(V)$
- a call C' that contains all the arguments that have not been used into the execution, hence useful for the next ones

$$\frac{SD = SD' \overset{N}{\texttt{argof}}\ SD'' \land \mathit{typeof}(\texttt{call}(C)) \sim SD' \land V = \mathit{execute}(\texttt{serve}(SD'), \texttt{call}(C))}{\texttt{serve}(SD') \cdot S \parallel S_S \parallel U_S \parallel B \cup SD' \cup \texttt{in\_call}(C, SD') \xrightarrow{\mathit{serve\_in\_call}} S \parallel U \parallel S_S \parallel U_S \parallel B \cup SD' \cup \texttt{in\_call}(N : T(V), C', SD'')} \quad \texttt{[SERVE-IN-CALL]}$$

`[LAST-IN-CALL]` operation plays the role of finisher of the computational chain. It is in fact the last operation to be executed.It is therefore in charge of executing the last service involved in the composition and providing the result to the user.

$$\frac{SD = SD' \overset{N}{\texttt{argof}}\ SD'' \land \mathit{typeof}(\texttt{call}(C)) \sim SD'' \land V = \mathit{execute}(\texttt{serve}(SD''), \texttt{call}(C))}{\texttt{serve}(SD'') \cdot S \parallel S_S \parallel U_S \parallel B \cup SD'' \cup \texttt{in\_call}(C, SD'') \xrightarrow{\mathit{last\_in\_call}} S \parallel \texttt{res}(V) \cdot U \parallel S_S \parallel U_S \parallel B \cup SD''} \quad \texttt{[LAST-IN-CALL]}$$

Prove

Operations `[POS-PROVE]` and `[NEG-PROVE]` are in charge of handling an exploration request `query`. To be triggered it requires that the request has previously been forwarded to the blackboard. They respectively show the positive case in which there is a service (or a composition of them) residing in the system that is able to fulfill it; and the negative one, where there is not.

$$\frac{\texttt{service}(Q) \sim SD \land Const = \mathit{prove}(Q, SD)}{\texttt{query}(Q) \cdot U \parallel S_S \parallel U_S \parallel B \cup SD \xrightarrow{\mathit{prove}} \texttt{res}(\top) \cdot U \parallel S_S \parallel U_S \parallel B \cup SD} \quad \texttt{[POS-PROVE]}$$

$$\frac{\nexists\ SD \in B : \texttt{service}(Q) \sim SD}{\texttt{query}(Q) \cdot U \parallel S_S \parallel U_S \parallel B \xrightarrow{\mathit{prove}} \texttt{res}(\bot) \cdot U \parallel S_S \parallel U_S \parallel B} \quad \texttt{[NEG-PROVE]}$$

The first rule describes the positive case, whilst the second the negative one. Preconditions show that to be in the positive case, there has to exist a service descriptor *SD* in the blackboard that matches the query request. If not

exists any service descriptor *SD* in the blackboard that matches the query request, then the negative response will be provided.
The match among the query request and a service descriptor *SD* is computed using the "∼" operator (see subsection 2.2.2 for the detailed definition).

Compose

Within the system, the composition of services is promoted and computed by means of the [COMPOSE] operation. It allows composing services through semantic reasoning on the service descriptors they provided on the blackboard.

$$\frac{SD = \texttt{service}(I,O) \wedge \exists\ (N:O) \in fringe(SD') \wedge SD'' = compose(SD,\ SD')}{S_S \parallel U_S \parallel B \cup SD \cup SD' \xrightarrow{compose} S_S \parallel U_S \parallel B \cup SD \cup SD' \cup SD''} \quad \texttt{[COMPOSE]}$$

Operation's preconditions define that the operation is triggered only when a simple service can be composed. The latter statement implicitly means that a service's output is included in the fringe of the one it will be composed with. Moreover, the correct execution of the *compose* function is included in the preconditions. The latter is simply a function that creates the actual match among two or more matchable services whenever it is possible (see section 2.2.2 for the detailed definition). During the operation the blackboard's state changes, being enriched with a new service descriptor (denoted by SD'' in the rule definition) that represents the new service created as result of the composition.

Decay

Finally, the operation [DECAY] is defined with the purpose of keeping the blackboard clean over time.

$$\frac{B' = B - compositions(B, SD)}{S_S \parallel U \parallel U_S \parallel B \cup SD \xrightarrow{\tau} S_S \parallel U \parallel U_S \parallel B'} \quad \texttt{[DECAY]}$$

As shown above, it holds no precondition, thus could be executed anytime. The existence of this operation grants the system the capability of cleaning out the blackboard from obsolete messages.

Label τ used for the operation describes a time related operation. Since there is no chance to express the time using the process algebra, the only way to express the operational semantic of `[DECAY]` is exploiting the concept of silent transition. In fact, τ defines a silent transition that occurs in the system and affects the state of the blackboard B. There is no specification about the time it occurs, just that after its occurrence the new state will not contain obsolete elements, that have been successfully removed during the transition.

2.2.2 Operators

The *typeof* function

The function is defined to cope with the need of retrieving the datatype of a call request. Retrieving the datatype allows to compare it with available services and evaluate if there exists one or more services able to fulfill the given request.

It is defined as follows:

$$typeof : \mathcal{L}(C) \rightarrow \mathcal{L}(SD)$$

It provides a service descriptor SD as output, starting from a call request $\texttt{call}(C)$ as input. Its semantic definition is:

$$typeof(\texttt{call}(X)) = \begin{cases} \texttt{service}(typeof(A), O) & \text{if } X = \texttt{call}(a_1 \dots a_n) \\ typeof(a_1) \dots typeof(a_n) & \text{if } X = a_1 \dots a_n \\ n : t & \text{if } X = n : t(v) \end{cases}$$

where

$$a_i = n_i : t_i(v_i)$$

Simply, the *typeof* function extracts the parameter's datatype and name and create a service descriptor SD that represents the request by means of the extracted values. The generated service descriptor will then be used to seek for an available and matchable service.

The $\sim$ operator

Operator “$\sim$” is in charge of evaluating if two service descriptors are matchable. It is defined as a relation between two service descriptors:

$$\sim\ \subseteq \mathcal{L}(SD) \times \mathcal{L}(SD)$$

Given two service descriptors, the operator evaluates if they are matchable or not, reasoning about their functional aspects (namely, datatype they present). The result is achieved exploiting the *semantic_match* predicate that embodies the semantic reasoning about datatype. However, it will not be defined within the model because it is meant to be application dependent.

$$\begin{aligned}
\texttt{service}(i_1 \dots i_n, o) \sim \texttt{service}(i'_1 \dots i'_n, o') &\Leftrightarrow i_j \sim i'_j\ \forall j = 1 \dots n \wedge o \sim o' \\
n : t \sim n : t' &\Leftrightarrow n = n' \wedge t \sim t' \\
t \sim t' &\Leftrightarrow \mathit{semantic\_match}(t, t')
\end{aligned}$$

$$SD \overset{N}{\texttt{argof}} SD' \sim SD'' \Leftrightarrow SD'' \sim SD \overset{N}{\texttt{argof}} SD'$$

$$\begin{aligned}
\texttt{service}(i_1 \dots i_n, o) \sim \texttt{service}(i'_1 \dots i'_m, o') \overset{N}{\texttt{argof}} SD \Leftrightarrow\ & n \geq m \wedge I = \{i_1 \dots i_n\} \wedge \\
& I' = \{i'_1 \dots i'_m\} \wedge \\
& I'' = I' \cap I \wedge \\
& I \equiv I' \equiv I'' \wedge \\
& \exists o''\ \texttt{service}(N : o', o'') \overset{o}{\sim} SD
\end{aligned}$$

$$\begin{aligned}
\texttt{service}(i_1 \dots i_n, o) \sim \texttt{service}(i'_1 \dots i'_m, o') \overset{N}{\texttt{argof}} SD \Leftrightarrow\ & n \geq m \wedge I = \{i_1 \dots i_n\} \wedge \\
& I' = \{i'_1 \dots i'_m\} \wedge \\
& I'' = I' \cap I \wedge \\
& I - I'' \neq I''' \wedge \\
& I''' = \{i'' \dots i''_{n-m}\} \wedge i_0 = N : o' \wedge \\
& \exists o''\ \texttt{service}(i'' \dots i''_{n-m}, i_0, o'') \overset{o}{\sim} SD
\end{aligned}$$

where

$$\begin{aligned} I \cap I' &= \{i : i \in I \wedge \exists i' \in I' : i \sim i'\} \\ i &\in \mathcal{L}(I) \\ o &\in \mathcal{L}(O) \equiv o \in \mathcal{L}(T) \end{aligned}$$

There could be two possible scenarios where the operator is used:

- both of the service descriptors are simple. By saying simple it is meant that they do not describe a composition

- one service descriptor is not simple, thus represents a composed service

The scenario in which both of them are composed cannot exist because the operator is meant to be used with a request (that is a simple service descriptor by definition) and a service.

The computational steps of the operator are defined in order to cope with both the scenarios. The basic scenario in which both service descriptors are simple is defined within the first three rows. Within the other part of the definition the rules in charge of computing the matching when the most articulated scenario occurs are shown.

Briefly, in the luckiest case, the service descriptors match when the number of their input parameter is the same, each of them has the same name and their datatype semantically match. In addition, the datatype of the request's output must semantically match with the service output datatype. On the other hand, during the most articulated scenario the matching occurs between a simple service descriptor and a composed one. Note that here it is supposed that the system is able to provide a correct composed service. Within this scenario, the matching relation holds only when there is a composed service whose input parameters set semantically match with the requested ones. It is exploited the $\overset{x}{\sim}$ operator in order to compute the matching among the required output and the output of the composed service.

The $\overset{x}{\sim}$ operator

During the matching computation, the $\overset{x}{\sim}$ is exploited in order to compute the semantic matching among the output parameters. In particular, it is used to explore the composed service syntax chain and compute the matching among the requested output and the output of the composed service. As the "$\sim$" operator, it is defined as a relation between two service descriptors.

$$\overset{x}{\sim} \subseteq \mathcal{L}(SD) \times \mathcal{L}(SD)$$

where

$$x \in \mathcal{L}(O)$$

As already mentioned the operator is used to test if the output of the composed service semantically matches with the requested output. It navigates the description of the composed service and test the aforesaid condition.

$$\texttt{service}(i_1 \dots i_n, o) \overset{x}{\sim} \texttt{service}(i'_1 \dots i'_n, x') \Leftrightarrow n \geq 0 \wedge i_j \sim i'_j \; \forall j = 1 \dots n \wedge x \sim x'$$

$$SD \overset{x}{\sim} SD' \overset{N}{\texttt{argof}} SD'' \Leftrightarrow \exists SD''' : SD''' \overset{x}{\sim} SD''$$

The *execute* function

Function *execute* is as a partial function, thus does not provide an answer for every possible input value that can be given. The execution of the function aims to the fulfillment of a *call* request. Supposing there is an available service that is able to fulfill a *call* request, this function is actually the subject in charge of performing the computation, fulfilling the request and providing an output value.
In the following lines are shown respectively the definition and the semantic of the function.

$$\textit{execute} : \mathcal{L}(S) \times \mathcal{L}(Req) \rightarrow \mathcal{L}(V)$$

The function accepts as input a set that is the Cartesian product between two elements, respectively belonging to the languages generated by the production set S and *Req*. As output, an element that belongs to the language generated by the production set V is provided.

$$execute(\texttt{serve}(SD), \texttt{call}(C)) = \begin{cases} \texttt{res}(V) & \text{if } typeof(C) \sim SD \wedge SD = \texttt{service}(I, O) \\ \emptyset & \text{otherwise} \end{cases}$$

According to the above semantic, this partial function exists, and it is able to provide an acceptable result, only when the first input matches with $\texttt{serve}(SD)$ and the second with $\texttt{call}(C)$. For all the other inputs the function is not defined, therefore not able to provide any result. Recursion is exploited to fulfill a request that can only be fulfilled by a composed service.

The *prove* function

Function *prove* is defined to perform the evaluation of an exploration query $\texttt{query}(Q)$. By means of using it, it is possible to evaluate whether a $\texttt{query}(Q)$ can be fulfilled or not.

$$prove : \mathcal{L}(Req) \times \mathcal{L}(SD) \rightarrow \mathcal{L}(Const)$$

Function's semantic is defined as follows:

$$prove(\texttt{query}(Q), SD) = \begin{cases} \top & \text{if } \texttt{service}(Q) \sim SD \\ \bot & \text{otherwise} \end{cases}$$

It only computes when the input is an exploration query $\texttt{query}(Q)$. The result expresses the capability of the system to fulfill or not the given request at that time. Note that the evaluation of the query is related to the state of the system at the time it occurs.

The *fringe* function

Function *fringe* is in charge of computing the fringe of a service descriptor. The fringe of a service descriptor *SD* is defined as the set whose elements are the inputs of *SD*.

$$fringe : \mathcal{L}(SD) \rightarrow \mathcal{L}(I)$$

The function is exploited in the service composition phase. During the latter, the fringe evaluation is necessary to reason about the composability of two services. In brief, a service S_1 can be composed with a service S_2 if and only S_1's output semantically matches with an element that is contained in the fringe of S_2.

$$fringe(SD) = \begin{cases} \{i_1 \dots i_n\} & \text{if } SD = \texttt{service}(I, O) \wedge \\ & \quad I = (i_1 \dots i_n) \\ fringe(SD') \cup fringe(SD'') - \{N : O\} & \text{if } SD = SD' \overset{N}{\texttt{argof}} SD'' \wedge \\ & \quad SD' = \texttt{service}(I, O) \end{cases}$$

The function performs a recursive computation. If the given input is a simple service descriptor, then the output is the set of its inputs. Instead, in case in which the given input is a composed service, the output is computed as the set that is the union of the fringes of the two services and from which it is removed the parameter on which the services composed.

The *compose* function

Self-composition of services is highly promoted within the system. Function *compose* encloses the composition policies and designs the bonding, creating a network of virtual services starting from the existing ones. It takes two service descriptors as input and provides a new service descriptor that is the composition of them.

$$compose : \mathcal{L}(SD) \times \mathcal{L}(SD) \rightarrow \mathcal{L}(SD)$$

As previously mentioned, the "compose" phase exploits the *fringe* function to compute the feasibility of the composition. When the composition is performed a new service descriptor is provided.

$$compose(SD, SD') = \begin{cases} SD \overset{N}{\texttt{argof}} SD' & \text{if } SD = \texttt{service}(I, O) \wedge \\ & \qquad \exists (N : O) \in fringe(SD') \\ \emptyset & \text{otherwise} \end{cases}$$

With abuse of notation, it is used

$$\exists (N : O) \in fringe(SD')$$

to express that $\exists e_f \in fringe(SD')$ for which the relation

$$O \sim e_f$$

holds true.
Function *compose* is grounded on the *fringe* function to perform the compositions. Basically, it creates a new composition among services S_1 and S_2 only when the output of S_1 semantically matches with an element that is member of the fringe set of S_2. The new composed service will be available on the blackboard, represented by a composed service descriptor of the form

$$SD_1 \overset{N}{\texttt{argof}} SD_2$$

with SD_1 and SD_2 defined as the service descriptors of S_1 and S_2 respectively.

The *compositions* function

As already mentioned within the blackboard a decay process is always ongoing, removing service descriptors considered obsoletes. However, there exists also a compose process that is trying to compose services whenever it is possible. The coexistence of the processes might lead the blackboard to have an inconsistent state. In fact, if a process is decaying, all the composition it is involved in should decay and consequently removed from the blackboard. Function *compositions* aims to identify all the composition in which a service descriptor SD is involved.

$$compositions : \mathcal{L}(B) \times \mathcal{L}(SD) \rightarrow \mathcal{L}(SD)$$

$$compositions(B, SD) = \begin{cases} SD' & \text{if } SD' = \ldots\ SD\ \ldots\ \wedge \\ & \qquad SD = \texttt{service}(I, O) \\ \emptyset & \text{otherwise} \end{cases}$$

where the notation

$$SD' = \ldots\ SD\ \ldots$$

stands for

exists at least a SD', either simple or composed, in which SD is contained

Chapter 3

Prototype

Within this chapter a basic prototype that only includes the core functionalities of the model is given. It has been developed using a Java-based light-weight Prolog implementation, namely tuProlog [18], `TuCSoN` [43, 46] and the `ReSpecT`$\mathbb{X}$ [12] language, built upon `ReSpecT` [44].

3.1 Service agents

As largely mentioned a *service agent* is an active entity that provides services in the system. It exploits the blackboard as coordination medium, publishing its services on it and consuming request messages in order to fulfill them.

Provided services' description is required to be meaningful for machines, therefore, a semantic definition should be provided. A semantic definition could be made by describing the service's functional aspects in terms of ontology's concepts. In fact, by means of using the concepts defined in an ontology it is possible to create a machine-readable representation and leverage it in order to perform semantic reasoning computations. However, since the ontologies are domain specific, finding which one (or ones) better fits to the cause is necessary. In case an ontology with exploitable concepts does not exist yet, an ad-hoc one must be coined.

3.1.1 Semantic definition

A service must be semantically defined, providing a machine-readable description of itself. In this case, a functional description such as input/output parameters is enough to be used within the model. Recalling the defined syntax, the term in which machine-readable data are stored is the service descriptor *SD*.

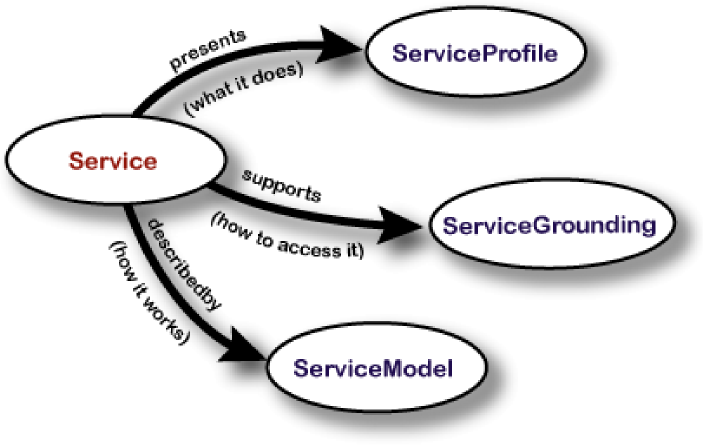

Figure 3.1: OWL-S Subontologies [36]

Regarding the description of a service using an ontology, it is exploited the OWL-S ontology (see figure 3.1. It defines an ontology wherewith is possible to semantically define a service, in all respects. To the aim of this work only the sub-ontology *ServiceProfile* is considered. However, further works shall not exclude the usage of the *ServiceModel*, since it is helpful to express behavioral aspects of the service that may lead to new reasoning processes and even more reliable service compositions. In the figure 3.2 it is shown in more detail the sub-ontology *ServiceProfile*.

A service is therefore defined in terms of its functional aspects by using the OWL-S ontology, as shown in the listing 3.1.

```
    <rdf:type>
      <owl:Restriction>
        <owl:onProperty rdf:resource="http://www.daml.org/
            services/owl-s/1.2/Profile.owl#hasInput"/>
        <owl:someValuesFrom rdf:resource="http://localhost/Book
            "/>
      </owl:Restriction>
      <owl:Restriction>
        <owl:onProperty rdf:resource="http://www.daml.org/
            services/owl-s/1.2/Profile.owl#hasInput"/>
        <owl:someValuesFrom rdf:resource="http://localhost/
            Writer"/>
      </owl:Restriction>
    </rdf:type>
    <rdf:type>
      <owl:Restriction>
        <owl:onProperty rdf:resource="http://www.daml.org/
            services/owl-s/1.2/Profile.owl#hasOutput"/>
        <owl:someValuesFrom rdf:resource="http://localhost/ISBN
            "/>
      </owl:Restriction>
    </rdf:type>
```

Listing 3.1: Semantic description of a service

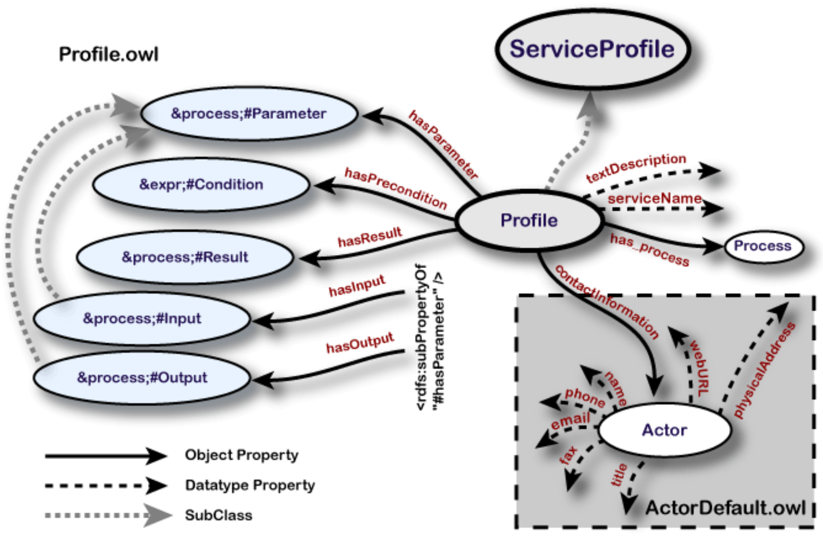

Figure 3.2: Top-level structure of the OWL-S profile [38]

3.1.2 From semantic to logic

The coordination model is grounded on logic-based computation. Due to this choice it is necessary to perform a conversion process, that provides a representation in the tuProlog syntax of the semantically defined service. However, within the prototype the component that performs the required conversion is missing. It is just assumed that a component with such capabilities exists and provides a representation as the one showed by listing 3.2. Briefly, `agent_name` corresponds to the identifier of the service (that is meant to be unique within the system), `input` to the accepted inputs and `output` to the provided output. Parameters names are chosen by the component in a functional manner. It can be observed that the logic representation is almost equivalent to the syntax defined by the model for a service descriptor *SD*. The latter may indeed defined as a composition of the `input` and `output`. Note that the predicate `id_gen(-Id)` provides a universal id based on the

```
init :- agent_name(N),
        node_address(A),
        node_port(P),
        acquire_acc(N, A, P),
        publish_sd,
        service_loop.

service_loop :- input(A),
               output(O),
               in(serve(call(Id, A, O))),
               execute(call(Id, A, O), V),
               out(res(Id, V)),
               service_loop.

agent_name(Id) :- id_gen(Id).
input([(title, book, T), (writer, writer, W)]).
output(isbn).
```

Listing 3.2: Logic description of a service agent

current timestamp.

3.2 User agents

It is expected that the *user agents* encapsulate within the request messages the metadata, namely semantic annotations, regarding the parameters. As defined in the previous chapter the requests are categorized in two classes: *query* and *call*. Note that by using a logic programming language, such as Prolog, the syntax defined in the formal model is almost directly usable within the implementation's code (as shown in listing 3.4).
A logic description of the user agent is shown in listing 3.3. As already mentioned the predicate `id_gen(-Id)` provides a universal id based on the current timestamp.

```
init :- agent_name(N),
        node_address(A),
        node_port(P),
        acquire_acc(N, A, P),

agent_name(Id) :- id_gen(Id).
```

Listing 3.3: Logic description of a user agent

```
% query(+Id, +Input, +Output)
input([(name, city)]).
output(kelvin).

% call(+Id, +Params, +Output)
parameters([(name, city, "Cesena")]).
output(kelvin).
```

Listing 3.4: Example of a query and call request using a logic description

3.3 Blackboard

The entity in charge of providing a coordination medium to the system is the *blackboard B*. It is also in charge of supporting the matching phase between a request and the available services and promoting self-composition among them. It is thus convenient to develop the blackboard as a tuple center. The latter shows all the properties requires by the described blackboard. To this purpose, TuCSoN and ReSpecT𝕏 are exploited, providing an engineered tuple space in which the core operations defined by the model are contained and accordingly triggered.
Moreover, the blackboard acts as container for the knowledge base regarding the ontologies. Since the computation grounds on the semantic reasoning among the concepts and it is performed using a logic approach, a logic representation of the ontology is required. Semantic reasoning processes are performed within the blackboard, therefore the presence of the knowledge allows computations to take place locally that may surely lead to more performing reasoning processes.

```
class(thing).

class(isbn).

class(city).

class(book).

class(person).
class(writer).
subclass(writer, person).

class(distance).
class(km).
class(miles).
subclass(km, distance).
subclass(miles, distance).

class(location).
class(coordGPS).
subclass(coordGPS, location).

class(library).
```

Listing 3.5: Logic description of the ontology hierarchy

3.3.1 Knowledge base

The knowledge base embodies the ontology hierarchy, hence the concepts defined by it. It aims to provide a logic representation of the hierarchy in order to be exploited during reasoning processes. Once again, within this prototype, the component capable of converting the hierarchy into a logic language is not provided due to its marginal role within the work. It is then assumed that, starting from a hierarchy such as the one shown in figure 4.6, the component is capable of providing a representation in a logic language such as 3.5. In [27, 34] such integration is argued. Note that in the above listing the subclass relation among all the concepts and the `class(thing)` is left out to shorten it.

3.4 Core operations

Core operations are developed by means of tuProlog predicates, exploited within reactions rules defined using the ReSpecTX language. For the sake of brevity, only the signatures of the predicates are shown. Their full implementation can be found at [1].

Service descriptor and request publication

Following listings show the operations performed during the publication phase of both a service descriptor (listing 3.6) and a call request (listing 3.7).

```
publish_sd :-
    agent_name(Id),
    input(I),
    output(O),
    out(service(Id, I, O)).
```

Listing 3.6: Operation [PUBLISH-SD]: publication of a service descriptor

```
publish_call :-
        agent_name(Id),
        parameters(A),
        output(O),
        id_gen(CallId),
        out(call(CallId, A, O)).
```

Listing 3.7: Operation [PUBLISH-CALL]: publication of a call request

Prove

Operations [POS-PROVE] and [NEG-PROVE] are implemented as reaction rules, shown in listing 3.8. It is defined the predicate prove(+SDquery) as the one in charge of computing the prove of the given exploration query. It exploits two more predicates, prove(+SDquery, +SimpleService) and

[1]https://bitbucket.org/ashleycaselli/lbc-semantic-self-composition

```
reaction to out query(Id, I, O) : completion, from_agent {
    in(query(Id, I, O)),
    if prove(service(Id, I, O)) then {
        out(res(Id, true))
    } else {
        out(res(Id, false))
    }
}
```

Listing 3.8: Operations [POS-PROVE] and [NEG-PROVE] implemented in ReSpecTX

prove(+SDquery, +ComposedService), that accept a second parameter whom represents a service descriptor, respectively simple and composed. The result of the computation is provided as a tuple in the form of res(Id, Const), with Const defined as a boolean value.

Serve

Within this section the reaction rules involved in the serve of a call request are shown. It is here assumed that by the time a request call reaches the tuple center, all the possible compositions among services have already been computed, hence using a dynamic deployment time approach for the service composition.

The listing 3.9 shows the implementation of the reaction rule that embodies the operations [SERVE-CALL] and [IN-CALL]. Consequently, in case the call request cannot be fulfilled by a simple service, the actual call is redesigned using the in_call syntax as new call request message. The handling of the calls is then performed through the [SERVE-IN-CALL] operation, shown in listing 3.10. The latter is triggered each time a in_call message reaches the tuple center and the service in charge of executing the actual call is not the last one in the composition chain, i.e. it is the header. The result of the service execution will be then modeled as a new parameter for the next in_call message, by means of the predicate input_set(+I, +Res, -NewSet), and attached to the it.

```
reaction to out call(Id, A, O) : completion, from_agent {
  typeof(call(Id, A, O), SDcall),
  if match(SDcall, service(Sname, I2, O2)) then {
    inp(call(Id, A, O)),
    out(serve(Sname, call(Id, A, O)))
  } else if match(SDcall, Scomp) {
    inp(call(Id, A, O)),
    out(in_call(call(Id, A, O), Scomp))
  } else {
    fail
  }
}
```

Listing 3.9: Operations [SERVE-CALL] and [IN-CALL] implemented in ReSpecTX

```
reaction to out in_call(call(Id, A, O), Scomp) : completion,
    internal {
  Scomp = [Sname, [argof, N]|T],
  rd(in_call(call(Id, A, O), Scomp)),
  out(serve(Sname, call(Id, A, O)))
}

reaction to out res(Id, V) : completion, from_agent {
  inp(res(Id, V)),
  Scomp = [_, [argof, _]|T].
  if inp(in_call(call(Id, A, O), Scomp)) then {
    input_set(A, res(V), NewSet),
    out(in_call(call(Id, NewSet, O), T))
  } else {
    out(res(Id, V))
  }
}
```

Listing 3.10: Operation [SERVE-IN-CALL] implemented in ReSpecTX

Once the service in charge of fulfilling the call is the last one in the composition chain, the operation [LAST-IN-CALL] is triggered and the result res(V) of the execution is given.

```
reaction to out in_call(call(Id, A, O), [Sname]) : completion,
    internal {
  inp(in_call(InId, call(Id, A, O), [Sname])),
  out(serve(Sname, call(Id, A, O)))
}
```

Listing 3.11: Operation [LAST-IN-CALL] implemented in ReSpecTX

Adopting a dynamic run time composition approach, the rules regarding the [SERVE-CALL] and [IN-CALL] operations would change as follows:

```
reaction to out call(Id, A, O) : completion, from_agent {
  typeof(call(Id, A, O), SDcall),
  if match(SDcall, service(Sname, I2, O2)) then {
    inp(call(Id, A, O)),
    out(serve(Sname, call(Id, A, O))),
  } else if match(SDcall, Scomp) {
    compose(service(S3, I3, O3), service(S4, I4, O4), Scomp),
    inp(call(Id, A, O)),
    out(in_call(call(Id, A, O), Scomp)).
  } else {
    fail
  }
}
```

Listing 3.12: Operations [SERVE-CALL], [IN-CALL] and [COMPOSE]: dynamic composition grounded on the run time approach

Compose

Within this section the service composition is argued. Both dynamic approaches are taken into account, namely deployment time and run time.

The composition operation [COMPOSE], shown in the listing 3.13, performs a dynamic composition at deployment time. The rule is therefore triggered each time a service is published in the tuple center. This solution is not suitable in those cases in which there exists an high number of services, since all the possible combinations of composition must be computed.

```
reaction to out service(Sname, I, O) : completion, from_agent {
    rd_all(service(Sname, I, O)) returns List,
    length(List, N),
    N > 2,
    compose(service(Sname, I, O), service(Sname2, I2, O2),
        SDcomp),
    out(SDcomp)
}
```

Conversely, the run time composition approach is more flexible and scalable. However, it introduces delays in the execution process, since the compositions are searched and made only when a request is received. Since it is executed during the serve of a call, the related code is shown in listing 3.12.

The predicate compose(+SD1, +SD2, -SD12) takes as input two service descriptors and provides a new service descriptor that is the composition of them. It is defined as a recursive predicate, thus is looking for all the possible compositions. The new composed services are then stored in the tuple center, in order to be available to the users.

Decay

Decay operation is developed as a periodic activity. The module to achieve the periodic behavior is already provided within the ReSpecTX Standard Library [2]. When each service is being published, a timestamp argument is attached to it. The periodic routine looks for those that have been pub-

[2]https://bitbucket.org/gciatto/respectx-standard-library/src

```
reaction to out decay(DT) {
  inp(decay(DT)),
  in_all(service(Id, I, O)),
  current_time(T),
  T - Id > DT,
  decay_compositions(Sid)
}
```

Listing 3.14: Operation `[DECAY]` implemented in `ReSpecTX`

lished at a timestamp t where

$$\mid t - currentTime \mid > \Delta t \tag{3.1}$$

(with Δt properly tuned).
If there is any, it is identified as obsolete and removed from the blackboard. The service who wish to continue providing its functionalities must perform another publish operation. Remember that not only the target service/s will be removed from the blackboard but also all the composed services in which it is/they are involved will be removed.

The execution of the periodic is performed as follows: (i) the tuple center emits the `start_periodic(Period, decay(DeltaT))` tuple to trigger the periodic emission of the `decay(DeltaT)` tuple; (ii) the tuple represents the activity to be performed once every `Period` milliseconds; (iii) the tuple center reacts by removing the services, according to the rule shown in listing 3.1, and with thsose all the compositions in which they are involved. In particular, the predicate `current_time(-T)` is used to retrieve the current time, expressed in milliseconds. Furthermore, all the composed services in which the decaying service is involved are identified by means of the predicate `compositions(+Sid, -List)`. It provides a list that contains all the composed services in which the service identified by `Sid` is involved. It is internally exploited by the predicate `decay_compositions(+Sid)` that is the one in charge of actually remove all the composed services from the blackboard, i.e. tuple center.

The remainder functions and predicates used within the rules are shown in the following listings. For the sake of brevity only their signatures are shown.

```
typeof(+Call, -SD).
```

Listing 3.15: Signature of the *typeof* function

```
match(+SDreq, +SDserv).
```

Listing 3.16: Signature of the $\sim$ operator

```
x_match(+SimpleSD, +SD, +X).
```

Listing 3.17: Signature of the $\overset{x}{\sim}$ operator

```
fringe(+SD, -Fringe).
```

Listing 3.18: Signature of the *fringe* function

```
execute(+Call, -Res).
```

Listing 3.19: The *execute* predicate

The matchmaking is performed by means of the `semantic_match(?T1, ?T2, ?Dist)` predicate. It is the predicate on which almost all the previous ones are grounded. The implementation used in this prototype is shown in the listing 3.20. It solely computes the semantic distance among two given concepts.

```
semantic_match(T, T1, D) :- semantic_match(T, T1, D, 0).
semantic_match(T, T, D, D2) :- exists(T), D is D2.
semantic_match(T, T1, D, D2) :- is_subclass(X, T),
                                D3 is D2+1,
                                semantic_match(X, T1, D, D3).

exists(A) :- class(A).
is_subclass(Class, Super) :- exists(Class),
                             exists(Super),
                             subclass(Class, Super).
```

Listing 3.20: Implementation of the *semantic_match* predicate using tuProlog

Chapter 4

Assessment

Within the chapter, the assessment of the model is argued. Several examples will be provided in order to prove the correctness and highlight the usefulness and the benefits of the designed model. Since the model aims to promote self-composition of services, examples mostly concerning the composition of services will be shown. Throughout the chapter holds the assumption that there exists a unique blackboard B that is always reachable by any agent connected to the node where B is situated.
In addition, it is assumed that used semantic matching relation is a predicate that is able to compute the semantic distance between two concepts and it is valid only when their distance $d \leq 1$ (in order to ease the examples). The relation among two concepts is therefore valid only when they are equivalent concept or the second subsumes the first one [47].

Note that the distance parameter d is an internal configuration of the blackboard, thus cannot be tuned by any external agent. The semantic matching relation is defined as $semantic\_match(t, t')$ and its computational process is abstracted away here.

4.1 A generic scenario

A generic scenario is given with the purpose of helping the reader to acquire the necessary confidence to clearly understand the used notation.

Reader that already feels confident with syntax's reading can skip the following pages and go to the section 4.2.

Let S_1 be a service that is represented by the service descriptor SD_1. Assume now that a [PUBLISH-SD] operation occurs, with $B = \emptyset$. Right after the execution the blackboard's state is changed to:

$$B = \{SD_1\}$$

Let SD_1 be defined as:

$$SD_1 = \texttt{service}(param : typeA, typeB)$$

where *typeA* and *typeB* are datatype mapped to concepts of an existing ontology (e.g. the one shown in the figure 4.1. S_1 is thus defined as a single service that accepts as input a *typeA* parameter and provides a *typeB* output. Let *Req* be an exploration request (query) defined as follows:

$$Req = \texttt{query}(Q) = \texttt{query}(param : typeA, typeB)$$

Assume now that a user agent forwards the latter request to the system. In this case, the [POS-PROVE] operation will be triggered because the following relation is positively evaluated.

$$\underbrace{\texttt{service}(param : typeA, typeB)}_{\texttt{service}(Q)} \sim \underbrace{\texttt{service}(param : typeA, typeB)}_{SD_1}$$

Briefly, the relation holds true because:

- the number of the parameters is equivalent; and
- the semantic relation *semantic_match* holds true by definition on both their input and output datatype (since they present the same datatype)

Suppose now that a hierarchy of datatype is defined as in figure 4.1. Let *Req* be an exploration request defined as follows:

$$Req = \texttt{query}(param : typeC, typeB)$$

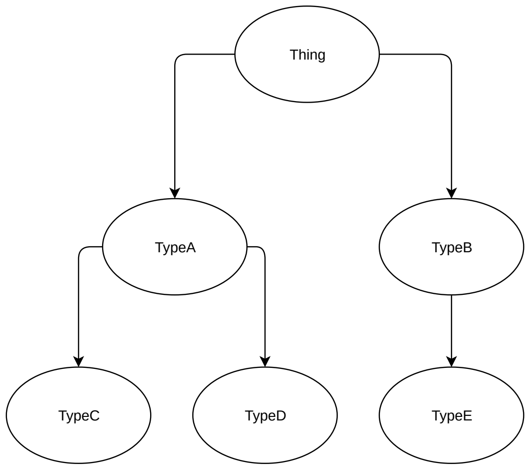

Figure 4.1: Generic datatype hierarchy

The relation

$$\underbrace{\texttt{service}(param : typeC, typeB)}_{\texttt{service}(Q)} \sim \underbrace{\texttt{service}(param : typeA, typeB)}_{SD_1}$$

will be once again positively evaluated because:

- the number of input parameters is equivalent
- *semantic_match(typeB, typeB)* holds true by definition
- *semantic_match(typeA, typeC)* holds true because *typeA* is the parent node of *typeC*, with distance $d = 1$

Therefore, the user request will positively evaluated by proving that the system holds the required capabilities to fulfill such a query. In particular, that the service capable to perform it is represented by the service descriptor SD_1.

4.2 Basic scenario

A practical example is shown to prove the correctness of the generic one. It exploits the hierarchy defined in figure 4.2 for the datatype definition.

Weather service

Imagine a user would like to know if exists a basic weather service, within the system, that given a city as input provides its current temperature. Since the user does not care about the unit of measure the temperature will be given, they do not provide any specification about it. The system is equipped with a service that is able to provide a temperature, measured in Celsius degrees in a given city. Intuitively, the request of the user may be fulfilled by the service even tough they offer different outputs.

Let define the service S_1 by means of the following service descriptor:

$$SD_1 = \texttt{service}(\mathit{name} : \mathit{City}, \mathit{Celsius})$$

It is here assumed that S_1 has already been published on the blackboard B, thus:

$$B = B' \cup \{SD_1\}$$

where B' represents the state of the blackboard B just before the publication of the service descriptor SD_1.
Let *Req* be a user exploration request, defined as follows:

$$\mathit{Req} = \texttt{query}(\mathit{name} : \mathit{City}, \mathit{Temperature})$$

By the time the request is sent, the prove computation will be triggered. The computational process will seek for a service descriptor SD_x, within B, for which the relation

$$\texttt{service}(\mathit{name} : \mathit{City}, \mathit{Temperature}) \sim SD_x$$

holds true.
In this case, the relation holds true when $SD_x \equiv SD_1$ because:

- the number of input parameters is equivalent
- $\mathit{semantic\_match}(\mathit{City}, \mathit{City})$ holds true by definition

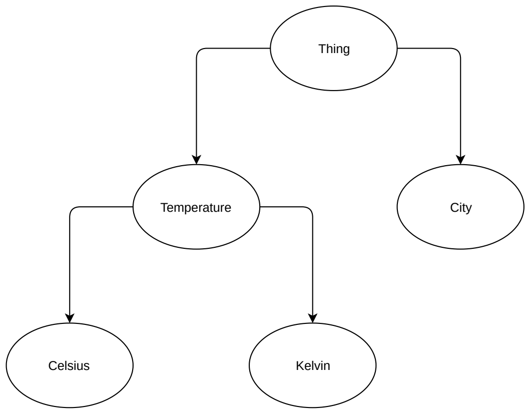

Figure 4.2: Weather datatype hierarchy

- $semantic\_match(Temperature, Celsius)$ holds true because the first term is the parent of the second one, with distance $d = 1$

Therefore the user's request will be positively evaluated, proving that the system is able to fulfill the given query. Indeed, if the user's request had been a call request with actual values as inputs, e.g.

$$\texttt{call}(name : City(Cesena), Temperature)$$

the system would have been able to fulfill it, exploiting the service S_1 since the following relation holds true.

$$typeof(call(name : City(Cesena), Temperature)) \sim SD_1$$

4.3 Composition scenario

In this section examples concerning the self-composition of services are provided. In order to ease the reader's reading and the examples presentation and explanation, the user's requests are always provided in the form of an exploration query.

4.3.1 Single-input services

Weather service

Following example is grounded on the previous weather service example (presented in the section 4.2).
Consider a user would like to know if there exists within the system a service that is able to provide the current temperature, expressed in Kelvin degrees, of a given city. The user request *Req* can be formally defined as:

$$Req = \texttt{query}(name : City, Kelvin)$$

The system is currently equipped with services S_1 and S_2 (see figure 4.3). If

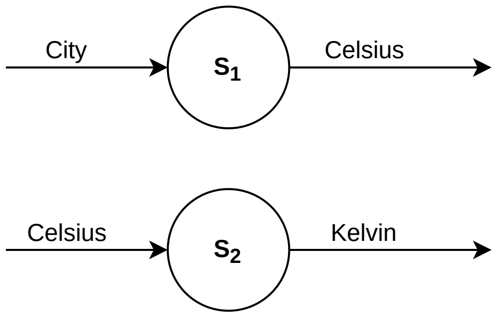

Figure 4.3: Weather single services

taken singularly, they are not able to wholly fulfill the request *Req*.
They are described by means of the following service descriptors:

$$SD_1 = \texttt{service}(name : City, Celsius)$$
$$SD_2 = \texttt{service}(temp : Celsius, Kelvin)$$

Blackboard B state is thereby:

$$B = B' \cup \{SD_1, SD_2\}$$

The blackboard notices that there are two services that might be composed in order to provide a new service with more functionalities. Since the following preconditions are valid:

$$\begin{aligned}
&SD_1 = \texttt{service}(name : City, Celsius)\ is\ simple && \wedge \\
&fringe(SD_2) = \{temp : Celsius\} && \wedge \\
&\exists(temp : Celsius) \in fringe(SD_2)
\end{aligned}$$

a [COMPOSE] operation will be triggered. Accordingly, the blackboard's state will be enriched with the new composed service descriptor SD_{12} that represents the virtual composed service S_{12} (see figure 4.4).

$$B = B' \cup \{SD_1, SD_2, SD_{12}\}$$

with

$$SD_{12} = SD_1 \overset{temp}{\texttt{argof}} SD_2 =$$
$$\texttt{service}(name : City, Celsius) \overset{temp}{\texttt{argof}} \texttt{service}(temp : Celsius, Kelvin)$$

The system capabilities have just been enhanced by creating a bonding

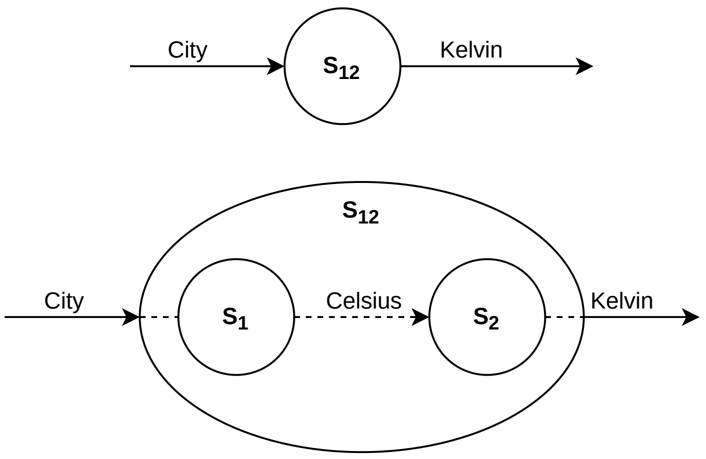

Figure 4.4: Weather services composed

among the available services S_1 and S_2.
By the time the request *Req* (previously defined) is sent, a prove computation

will be triggered. Since there is no simple service which makes the relation

$$\texttt{service}(name : City, Kelvin) \sim SD_x$$

to be valid, the matchable SD_x has to be sought among the composed services. Computational steps of the "$\sim$" operator follow:

$$\begin{aligned}
&\texttt{service}(name : City, Kelvin) \sim \overbrace{\texttt{service}(name : City, Celsius)}^{SD_1}\ \overset{N}{\texttt{argof}}\ SD_y \\
&\Updownarrow \\
&n = 1 \geq m = 1 \wedge \\
&I = \{name : City\} \wedge \\
&I' = \{name : City\} \wedge \\
&I'' = I' \cap I \wedge \\
&I \equiv I' \equiv I'' \wedge \\
&\exists o'' \texttt{service}(N : Celsius, o'') \overset{Kelvin}{\sim} SD_y
\end{aligned}$$

$$\begin{gathered}
\texttt{service}(N : Celsius, o'') \overset{Kelvin}{\sim} \overbrace{\texttt{service}(i_1 \dots i_n, Kelvin)}^{SD_y} \\
\equiv \\
\texttt{service}(N : Celsius, Kelvin) \overset{Kelvin}{\sim} \underbrace{\texttt{service}(temp : Celsius, Kelvin)}_{SD_y} \\
\Updownarrow \\
Celsius \sim Celsius \wedge Kelvin \sim Kelvin
\end{gathered}$$

Computational steps prove that exists a service, denoted as SD_y, that satisfies the relation. In this case, it can also be observed that $SD_y = SD_2$, thereby the user's request may be fulfilled. In fact, it is proved that it is exactly the composition of S_1 and S_2 that produces a new service S_{12} that owns the needed capabilities to fulfill the request.

4.3.2 Multi-input services

In the following scenario it is shown how the composition process among multi-input services occurs. However, single input services are also involved in the composition process to facilitate the formal proving. The composition process will lead to the creation of new services that might be exploited to fulfill a user agent request. To this purpose, it is shown how a composed service is selected by the blackboard as the one in charge of fulfilling a user request.

Book seeking

Scenario involves a user agent that would like to know if there is a library in a given radius far away from a city in which a specified book is available. User's request Req is defined as:

$$\begin{aligned} Req = \texttt{query}(&title : Book, name : Person, \\ &city : City, radius : Distance, Library) \end{aligned}$$

Let the blackboard B be:

$$B = B' \cup \{SD_1, SD_2, SD_3\}$$

where SD_1, SD_2 and SD_3 describe three services, respectively S_1, S_2 and S_3 (see figure 4.5). Service descriptors' definition is the following:

$$\begin{aligned} &SD_1 = \texttt{service}(title : Book, writer : Writer, ISBN) \\ &SD_2 = \texttt{service}(name : City, CoordGPS) \\ &SD_3 = \texttt{service}(code : ISBN, location : CoordGPS, distance : Km, Library) \end{aligned}$$

The aforementioned services exploit the concepts hierarchy shown in figure 4.6 to express their input/output datatype.

In this case, a `[COMPOSE]` operation is triggered, trying to compose the services previously described. In fact, it can be seen from the figure 4.5 that they share parameters on which a composition may occur. The composition

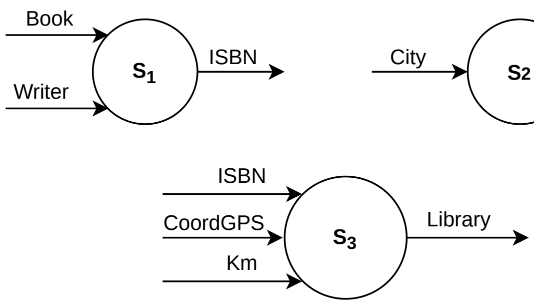

Figure 4.5: Book seeking services

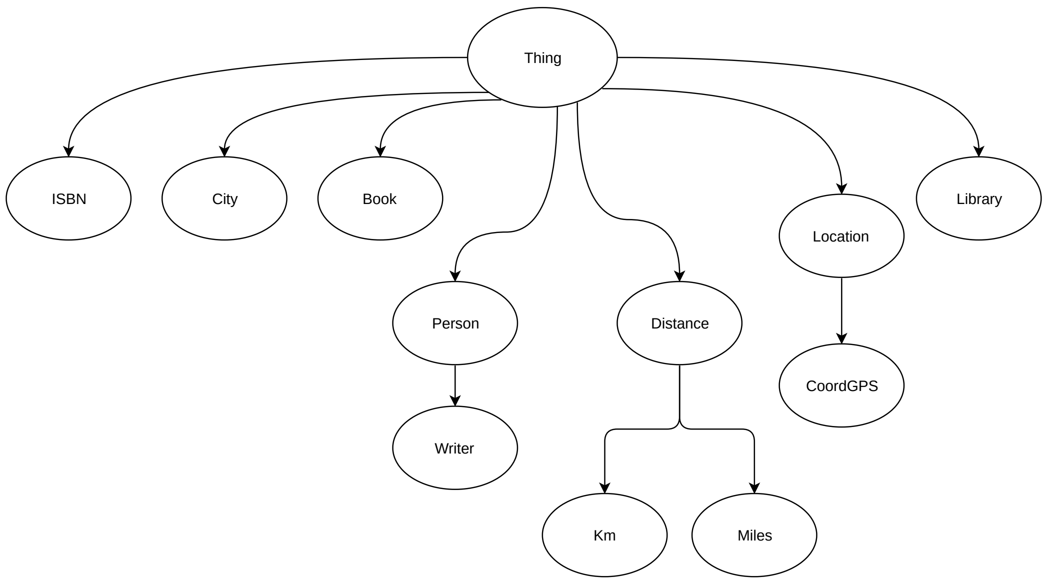

Figure 4.6: Book seeking datatype

process will end creating a new composed service, namely S_{123}, through the following computational steps:

$$SD_2 = \texttt{service}(city : City, CoordGPS) \ \textit{is simple} \quad \wedge$$
$$fringe(SD_3) = \{code : ISBN, location : CoordGPS, distance : Km\} \quad \wedge$$
$$\exists (location : CoordGPS) \in fringe(SD_3)$$

$$\Downarrow$$

$$SD_{23} = SD_2 \overset{location}{\texttt{argof}} SD_3$$

$$\begin{aligned}
&SD_1 = \texttt{service}(title : Book, writer : Writer, ISBN)\ is\ simple && \wedge\\
&fringe(SD_{23}) = \{city : City, code : ISBN, distance : Km) && \wedge\\
&\exists(code : ISBN) \in fringe(SD_{23})
\end{aligned}$$

$$\Downarrow$$

$$SD_{123} = SD_1 \overset{code}{\texttt{argof}} SD_2 \overset{location}{\texttt{argof}} SD_3$$

A graphical representation of the composed service S_{123} is shown in figure 4.7. At the end of the `[COMPOSE]` operation computation, the blackboard will be enriched with the new composed service descriptor SD_{123}, becoming:

$$B = B' \cup \{SD_1, SD_2, SD_3, SD_{123}\}$$

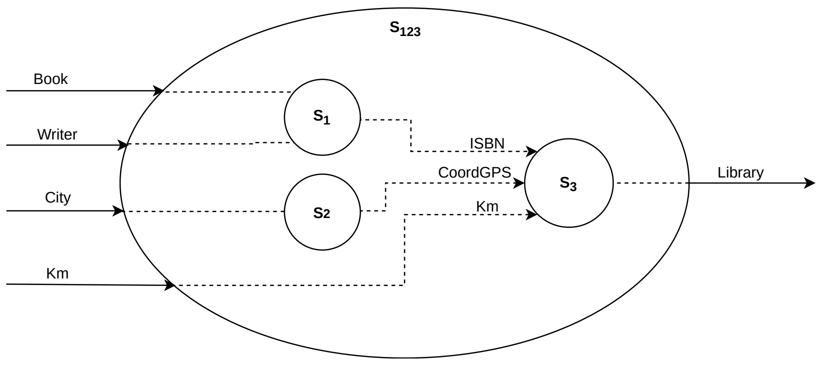

Figure 4.7: Book seeking service composition

It is here assumed that by the time the request *Req* is sent, the composed service S_{123} is already existing within the blackboard B. Due to this, a prove process will be immediately started, proving that the new composed service

SD_{123} is the proper candidate to fulfill the request by the following steps:

$$\begin{aligned}
&Q = title : Book, name : Person, city : City,\\
&\quad radius : Distance, Library\\
&\texttt{service}(Q) \sim SD_x\\
&\texttt{service}(Q) \sim \texttt{service}(title : Book, writer : Writer, ISBN) \overset{N}{\texttt{argof}} SD_y
\end{aligned}$$

$$\Updownarrow$$

$$\begin{aligned}
&n = 4 \geq m = 2 && \wedge\\
&I = \{title : Book, name : Person, city : City, radius : Distance\} && \wedge\\
&I' = \{title : Book, writer : Writer\} && \wedge\\
&I'' = I' \cap I && \wedge\\
&I - I'' \neq I''' && \wedge\\
&I''' = \{city : City, radius : Distance\} && \wedge\\
&i_0 = code : ISBN && \wedge\\
&\exists o'' \texttt{service}(city : City, radius : Distance, code : ISBN, o'') \overset{Library}{\sim} SD_y
\end{aligned}$$

$$\begin{aligned}
&\texttt{service}(city : City, radius : Distance,\\
&\quad code : ISBN, o'') \overset{Library}{\sim} SD' \overset{N}{\texttt{argof}} SD''
\end{aligned}$$

$$\Updownarrow$$

$$\underbrace{\texttt{service}(city : City, radius : Distance, code : ISBN, o'')}_{SD_z} \overset{Library}{\sim} SD_{23}$$

According to the defined rules, it can be observed that the service descriptor that makes the relation to be true is

$$SD_y = SD_{23}$$

since

- the service descriptor SD_{23} of composite service S_{23} has the same number of the parameters of SD_z, and
- for each input/output the *semantic_match* relation holds true

Conclusions

In this dissertation a semantic approach to self-composition of services situated in a multi-agent systems (MAS) environment has been provided. The approach presents a new coordination model grounded on logic-based coordination ones, which promotes and supports self-composition of services. The model has been conveyed using a formal approach, exploiting the extended Backus-Naur form (EBNF) notation for its syntax and a labeled transition system (LTS) for the operational semantics. A prototype containing the core operations has also been implemented using tuProlog, a Java-based lightweight implementation of Prolog, `TuCSoN` and the `ReSpecT`$\mathbb{X}$ language.

The work presented in this dissertation has a strong relation to the field of automatic service composition [4, 41, 29]. While approaches to automatic composition are mostly syntactic, other techniques have been devised to deal with semantics [33] and dynamic scenarios such as pervasive systems [31].

Future developments might be performed in many directions. Regarding the model, it may be extended in order to support the description of the services under several aspects, such as non-functional and behavioral. The latter would be an high valuable asset for the matchmaking process, leading it to the creation of more accurate matches. The composition process will also be able to reason about service's behaviors defined in a formal fashion, therefore leading to more reliable compositions. Further works might lead towards the adoption of NLP techniques to automatically annotate the user's requests with ontologies [50], thus not requiring the user to be in possess of the semantic knowledge to be attached to the request. Moreover, within the model there is a lack of primitives wherewith manage quality of service (QoS) aspects, especially concerning the matchmaking policies.

Future works might also involve the coordination among multiple blackboard and/or nodes, the merging of the different knowledge bases (KB) using semantic matching approaches (such as [23, 24, 25]) in order to support cross-domain ontologies and adopting alternative algorithms to improve the computation on which the service composition processes rely (i.e. [51]).

Bibliography

[1] A. Abuarafah, H. Mohammed, and O. Khozium. Agent Vs Object with an in-depth insight to Multi-Agent Systems. *International Journal of Engineering Science*, vol.4, 2013.

[2] G. Antoniou and F. VanHarmelen. *A Semantic Web Primer*. MIT Press, Cambridge, MA, USA, 2004.

[3] B. Benatallah, M. Dumas, M.-C. Fauvet, and F. A. Rabhi. *Towards Patterns of Web Services Composition*, pages 265–296. Springer London, London, 2003.

[4] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella. Automatic Composition of Transition-based Semantic Web Services with Messaging. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 613–624. VLDB Endowment, 2005.

[5] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, may 2001.

[6] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web Services Architecture. World Wide Web Consortium, Note NOTE-ws-arch-20040211, 2004.

[7] R. Calegari, E. Denti, S. Mariani, and A. Omicini. Logic Programming as a Service. *Theory and Practice of Logic Programming*, 18(5-6):846–873, jun 2018.

[8] Y. Charif and N. Sabouret. An Overview of Semantic Web Services Composition Approaches. *Electronic Notes in Theoretical Computer Science*, 146(1):33–41, jan 2006.

[9] P. Ciancarini. Distributed programming with Logic Tuple Spaces. *New Generation Computing*, 12(3):251–283, jun 1994.

[10] P. Ciancarini. Coordination Models and Languages as Software Integrators. *ACM Computing Surveys*, 28(2):300–302, jun 1996.

[11] P. Ciancarini, A. Omicini, and F. Zambonelli. Multiagent System Engineering: the Coordination Viewpoint. In *Intelligent Agents VI. Agent Theories, Architectures, and Languages*, pages 250–259, 2000.

[12] G. Ciatto, S. Mariani, and A. Omicini. *Programming the Interaction Space Effectively with $$\texttt {ReSpecT}\mathbb {X}$$*, pages 89–101. Springer International Publishing, Cham, 2018.

[13] G. David and C. Naicholas. Coordination languages and their significance. *Communications of the ACM*, 35(2):97 – 107, feb 1992.

[14] F. L. De Angelis and G. Di Marzo Serugendo. Logic fragments: Coordinating entities with logic programs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9952 LNCS:589–604, 2016.

[15] F. L. De Angelis, G. Di Marzo Serugendo, D. Buchs, M. Massink, and A. Omicini. *Centre Universitaire d'Informatique Doctoral Program in Information Systems A Logic-Based Coordination Middleware for Self-Organising Systems: distributed reasoning based on many-valued logics.* PhD thesis.

[16] F. L. De Angelis, J. L. Fernandez-Marquez, and G. Di Marzo Serugendo. Self-composition of Services in Pervasive Systems: A Chemical-Inspired Approach. In G. Jezic, M. Kusek, I. Lovrek, R. J. Howlett, and L. C. Jain, editors, *Agent and Multi-Agent Systems: Technologies and Applications*, pages 37–46, Cham, 2014. Springer International Publishing.

[17] E. Denti, A. Natali, A. Omicini, and M. Venuti. *Logic Tuple Spaces for the Coordination of Heterogenous Agents*, pages 235–248. Springer Netherlands, Dordrecht, 1996.

[18] E. Denti, A. Omicini, and A. Ricci. tuProlog: A Light-Weight Prolog for Internet Applications and Infrastructures. In *Practical Aspects of Declarative Languages. 3rd International Symposium (PADL 2001), Las Vegas, Nevada, March 11–12, 2001 Proceedings*, volume 1990, pages 184–198. Springer-Verlag, 2001.

[19] G. Di Marzo Serugendo, N. Abdennadher, H. Ben Mahfoudh, F. L. De Angelis, and R. Tomaylla. Spatial edge services. In *2017 Global Internet of Things Summit (GIoTS)*, pages 1–6. IEEE, jun 2017.

[20] J. Domingue, L. Cabral, F. Hakimpour, D. Sell, and E. Motta. IRS-III: A platform and infrastructure for creating WSMO-based semantic web services. 2004.

[21] D. Fensel and C. Bussler. The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137, jun 2002.

[22] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, jan 1985.

[23] F. Giunchiglia and P. Shvaiko. Semantic matching. *Knowledge Engineering Review*, 18(3):265–280, 2003.

[24] F. Giunchiglia, P. Shvaiko, and M. Yatskevich. Semantic Schema Matching. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, pages 347–365, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[25] F. Giunchiglia, M. Yatskevich, and P. Shvaiko. Semantic Matching: Algorithms and Implementation. In S. Spaccapietra, P. Atzeni, F. Fages, M.-S. Hacid, M. Kifer, J. Mylopoulos, B. Pernici, P. Shvaiko, J. Trujillo,

and I. Zaihrayeu, editors, *Journal on Data Semantics IX*, pages 1–38, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[26] D. Greenwood and M. Calisti. Engineering web service - agent integration. In *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583)*, volume 2, pages 1918–1925. IEEE, 2004.

[27] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs. In *Proceedings of the twelfth international conference on World Wide Web - WWW '03*, page 48, New York, New York, USA, 2003. ACM Press.

[28] F. Hakimpour, T. Payne, L. Cabral, J. Domingue, and E. Motta. Approaches to Semantic Web Services: an Overview and Comparisons. pages 225–239. 2010.

[29] S. Hu, V. Muthusamy, G. Li, and H.-A. Jacobsen. Distributed automatic service composition in large-scale systems. In *Proceedings of the second international conference on Distributed event-based systems - DEBS '08*, page 233, New York, New York, USA, 2008. ACM Press.

[30] D. Isern, D. Sánchez, and A. Moreno. Organizational structures supported by agent-oriented methodologies. *Journal of Systems and Software*, 84(2):169–184, feb 2011.

[31] S. Kalasapur, M. Kumar, and B. A. Shirazi. Dynamic Service Composition in Pervasive Computing. *IEEE Transactions on Parallel and Distributed Systems*, 18(7):907–918, jul 2007.

[32] M. Ketel. Integration of Software Agent Technologies and Web Services. 2, 2009.

[33] S. Kona, A. Bansal, and G. Gupta. Automatic Composition of SemanticWeb Services. In *IEEE International Conference on Web Services (ICWS 2007)*, pages 150–158. IEEE, jul 2007.

[34] L. Laera, V. Tamma, T. Bench-Capon, and G. Semeraro. SweetProlog: A System to Integrate Ontologies and Rules. In G. Antoniou and H. Boley, editors, *Rules and Rule Markup Languages for the Semantic Web*, pages 188–193, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[35] A. L. Lemos, F. Daniel, and B. Benatallah. Web Service Composition: A Survey of Techniques and Tools. *ACM Computing Surveys*, 48(3):1–41, dec 2015.

[36] D. Martin, 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 Memb. Submiss.*, 22, 2004.

[37] D. Martin, M. Burstein, O. Lassila, M. Paolucci, T. Payne, and S. Mcilraith. Describing web services using OWL-S and WSDL. 2003.

[38] D. Martin, M. Burstein, D. McDermott, S. McIlraith, M. Paolucci, K. Sycara, D. L. McGuinness, E. Sirin, and N. Srinivasan. Bringing Semantics to Web Services with OWL-S. In *World Wide Web*, volume 10, pages 243–277, aug 2007.

[39] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, and D. Bringing Semantics to Web Services: The OWL-S Approach. *First International*, 3387:26 – 42, 2004.

[40] S. McIlraith, T. Son, and Honglei Zeng. Semantic Web services. *IEEE Intelligent Systems*, 16(2):46–53, mar 2001.

[41] N. Milanovic and M. Malek. Current solutions for Web service composition. *IEEE Internet Computing*, 8(6):51–59, nov 2004.

[42] E. Motta, J. Domingue, L. Cabral, and M. Gaspari. IRS–II: A Framework and Infrastructure for Semantic Web Services. In D. Fensel, K. Sycara, and J. Mylopoulos, editors, *The Semantic Web - ISWC 2003*, pages 306–318, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[43] A. Omicini. On the semantics of tuple-based coordination models. In *Proceedings of the 1999 ACM symposium on Applied computing - SAC '99*, pages 175–182, New York, New York, USA, 1999. ACM Press.

[44] A. Omicini. Formal ReSpecT in the A&A Perspective. *Electronic Notes in Theoretical Computer Science*, 175(2):97–117, jun 2007.

[45] A. Omicini and E. Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, nov 2001.

[46] A. Omicini and F. Zambonelli. Tuple Centres for the Coordination of Internet Agents. In *1999 ACM Symposium on Applied Computing (SAC'99)*, pages 183–190, New York, New York, USA, 1999. ACM Press.

[47] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, ISWC '02, pages 333–347, Berlin, Heidelberg, 2002. Springer-Verlag.

[48] G. A. Papadopoulos and F. Arbab. Coordination Models and Languages. *Advances in Computers*, 46(C):329–400, 1998.

[49] L. Ribeiro, J. Barata, and A. Colombo. MAS and SOA: A Case Study Exploring Principles and Technologies to Support Self-Properties in Assembly Systems. In *2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 192–197. IEEE, oct 2008.

[50] J. Saias and P. Quaresma. A methodology to create legal ontologies in a logic programming information retrieval system. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3369 LNAI, pages 185–200, 2005.

[51] G. Shu, O. F. Rana, N. J. Avis, and C. Dingfang. Ontology-based semantic matchmaking approach. *Advances in Engineering Software*, 38(1):59–67, jan 2007.

[52] M. ter Beek, A. Bucchiarone, and S. Gnesi. Formal methods for service composition. *Annals of Mathematics, Computing & Teleinformatics*, 1:1–10, 2007.

[53] M. Viroli and M. Casadei. Chemical-inspired self-composition of competing services. In *Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10*, SAC '10, page 2029, New York, New York, USA, 2010. ACM Press.

Acknowledgments

This thesis is the result of a joint project between the University of Bologna (Italy) and the University of Geneva (Switzerland), where I spent three months as a visiting researcher.

Firstly, I would like to express my sincere gratitude to Andrea Omicini, who supervised me in the past six months, and to Giovanni Ciatto, for his continuous support while working on the dissertation. Special thanks to Giovanna Di Marzo Serugendo, who supervised me during the time I spent at the University of Geneva, and to all the people I have met there that helped me and contributed to this dissertation.

Besides, I would like to thank my fellow students for the time we have spent together in these years.

I am deeply grateful to my family, especially to my parents and siblings, for supporting me throughout my life.