A SEMANTIC FRAMEWORK FOR EVENT-DRIVEN SERVICE COMPOSITION

A DISSERTATION IN
Computer Science
and
Telecommunications and Computer Networking

Presented to the Faculty of the University
of Missouri Kansas City in partial fulfillment of
the requirements for the degree

DOCTOR OF PHILOSOPHY

by

SOURISH DASGUPTA

M.C.A, West Bengal University of Technology, 2006

Kansas City, Missouri

2011

A SEMANTIC FRAMEWORK FOR EVENT-DRIVEN SERVICE COMPOSITION

Sourish Dasgupta, Candidate for the Doctor of Philosophy Degree

University of Missouri - Kansas City, 2011

ABSTRACT

Service Oriented Architecture (*SOA*) has become a popular paradigm for designing distributed systems where loosely coupled *services* (i.e. computational entities) can be integrated seamlessly to provide complex *composite services*. Key challenges are discovery of the required services using their formal descriptions and their coherent composition in a timely manner. Most service descriptions are written in XML-based languages that are syntactic, creating linguistic ambiguity during service matchmaking. Furthermore, existing models that implement SOA have mostly middleware-controlled synchronous request/reply-based runtime binding of services that incur undesirable service latency. In addition, they impose expensive state monitoring overhead on the middleware. Some newer event-driven models introduce asynchronous publish/subscribe-based event notifications to consumer applications and services. However, they require an event-library that stores definitions of all possible system events, which is impractical in an open and dynamic system.

The objective of this study is to efficiently address on-demand consumer requests with minimum service latency and maximum consumer utility. It focuses on semantic event-driven service composition. For efficient semantic service discovery, the dissertation proposes a novel service learning algorithm called *Semantic Taxonomic Clustering* (*STC*).

The algorithm utilizes semantic service descriptions to cluster services into functional categories for pruning search space during service discovery and composition. STC utilizes a dynamic bit-encoding algorithm called *DL-Encoding* that enables linear time bit operation-based semantic matchmaking as compared to expensive reasoner-based semantic matchmaking. The algorithm shows significant improvement in performance and accuracy over some of the important service category algorithms reported in the literature. A novel user-friendly and computationally efficient query model called *Desire-based Query Model* (DQM) is proposed for formally specifying service queries. STC and DQM serve as the building block for the dual framework that is the core contribution of this dissertation: (i) centralized *ALNet* (*Activity Logic Network*) platform and (ii) distributed agent-based *SMARTSPACE* platform. The former incorporates a middleware controlled service composition algorithm called *ALNetComposer* while the latter includes the *SmartDeal* purely distributed composition algorithm. The query response accuracy and performance were evaluated for both the algorithms under simulated event-driven SOA environments. The experimental results show that various environmental parameters, such as domain diversity and scope, size and complexity of the SOA system, and dynamicity of the SOA system, significantly affect accuracy and performance of the proposed model. This dissertation demonstrates that the functionality and scalability of the proposed framework are acceptable for relatively static and domain specific environments as well as large, diverse, and highly dynamic environments. In summary, this dissertation addresses the key design issues and problems in the area of asynchronous and pro-active event-driven service composition.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Graduate Studies, have examined a dissertation titled "A Semantic Framework for Event-driven Service Composition" presented by Sourish Dasgupta, candidate for the Doctor of Philosophy degree, and certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Yugyung Lee, Ph.D., Committee Chair
School of Computing and Engineering

Vijay Kumar, Ph.D.
School of Computing and Engineering

Deendayal Dinakarpandian, Ph.D., M.D.
School of Computing and Engineering

Appie Van de Liefvoort, Ph.D.
School of Computing and Engineering

Deepankar Medhi, Ph.D.
School of Computing and Engineering

Baek-Young Choi, Ph.D.
School of Computing and Engineering

CONTENTS

ILLUSTRATIONS

Figure Page

TABLES

Table Page

LIST OF ABBREVIATIONS

ALNet	Activity Logic Network
DAML	DARPA Agent Markup Language
OIL	Ontology Interchange Layer
OWL	Web Ontology Language
OWL-S	Web Ontology Language - Service
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
URI	Uniform Resource Identifier
WWW	World Wide Web
JADE	Java Agent Development Environment
SA	Service Agent
SHA	Service Helper Agent
UA	User Agent
DA	Directory Agent
BA	Blackboard Agent

ACKNOWLEDGEMENTS

I gratefully acknowledge my advisor Dr. Yugyung Lee for her continuing guidance and support during the course of this research. She has been a constant source of inspiration over my years in this department. Without her the thesis would never have been possible. I also sincerely thank my PhD committee members Dr. Deendayal Dinakarpandian, Dr. Vijay Kumar, Dr. Appie Van de Liefvoort, Dr. Deepankar Medhi, and Dr. Baek-Young Choi for taking time off their busy schedule to render their thoughts, advice, and critical comments regarding my research and for their continuous encouragement throughout my program. I would also like to show my special gratitude to the UMKC Preparing for Future Faculty Fellowship (PFF) program for providing generous financial support for the last three years.

I am also indebted to the ARTISAN and the SMARTSPACE project teams for their sleepless nights of coding and testing and my abnormally frequent demands for tea. Thanks a lot to Nitin Mamillapally, Satish Bhat, Jia Zhang, Sourav Jana, Teja Mylavarapu, and Sudeep Maity for bearing with me. I love you all.

Finally I would like to thank my wife Tina for happily enduring my negligence over the last three crazy years and caring for me so much during the last few months of my dissertation writing. I don't think I can ever parallel her love for me.

The views and conclusions contained herein are those of the author's and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the University of Missouri - Kansas City.

CHAPTER 1

INTRODUCTION

1.1 Research Motivation

Over the past decade Service Oriented Architecture (SOA) has evolved into a generic design paradigm widely adopted in distributed systems such as WWW cloud [1], P2P based systems, and pervasive systems [2]. The underlying principles of SOA draw inspiration from the earlier distributed component based models such as Microsoft's DCOM [3] and OMG's CORBA [4]. Both these models aimed at integrating independent software components over network so as to provide a unified service to a consumer. This problem of providing a complex service from independent decoupled software components is called *service composition*. Often the software components would be written in very different languages and hosted in completely heterogeneous platforms. The challenge was to interface the software components in a manner such that they can call each other over RPC. For this purpose most models invented their proprietary interface languages (e.g., IDL for CORBA) that mapped object messages from a calling component to the native language of the receiving component.

However, there were several drawbacks of component based models that led to the evolution of SOA. One of the most important drawbacks was Internet firewalls that prevented inter-component communications. Another problem was that programmers had to be skilled in interface languages and complex mappings. The third problem (and perhaps as far as the motivation of this research goes the most important one) was the inability of the system as a whole to dynamically call up components as and when required. Although

software components were independent for providing a complex integrated service they had to be tied up programmatically at the client front-end application so that the RPC calls could take place. In other words, service composition is static in these models. In an evolving dynamic and non-deterministic system where old components are modified and new components are added in and where competing components exist (with their individual cost and QoS) static service composition cannot provide an optimal solution. We need an architecture that can retrieve required software components according to a consumer demand and then assemble them on-the-fly by selecting the best components in terms of cost and QoS in order to provide the desired complex service. Studying and solving this problem, known as *dynamic service composition*, is going to be the core of this dissertation.

In order to understand how SOA helps us to provide an architectural framework for formalizing and solving dynamic service composition (from now onwards will be called simply as service composition) let us look at the popular general definition of SOA given by OASIS [5]:

"*Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains*".

As per this definition we see that the components within an SOA-based system may be not be just software but can also include any computational capability ranging from devices, agents, hardware resources (e.g., CPU, memory, etc), dynamic linking libraries (DLL), databases, and network resources. We also see from the definition that such capabilities can have different ownerships. This is a very important feature as it demands a method of communication that is not proprietary and limited within an Internet firewall. The

idea behind this definition becomes clearer from the following definition given by Nickull [6]:

"*SOA is an architectural paradigm for components of a system and interactions or patterns between them such that a component offers a service that waits in a state of readiness and other components may invoke the service in compliance with a service contract*".

The above definition emphasizes two important facets of SOA: (i) components as 'ready-to-execute' *services* and (ii) discovering a set of inter-service behavioral interactions or patterns (i.e. service composition). These two aspects of SOA are very clearly incorporated in the widely referenced definition of *services* given by Papazoglou and Georgakopoulos [7] which states:

"*Services are self-describing, open components that support rapid, low-cost composition of distributed applications*".

To summarize, SOA principles provides a framework to model systems where:

- Service hosting platforms are distributed and heterogeneous.
- Service providers are independent stakeholders.
- Services are either loosely coupled (minimum mutual dependency) or completely independent (no mutual awareness).
- Services include all kinds of computational entities.
- Services are accessible over Internet firewalls.
- Services are reusable as participants in satisfying multiple complex consumer demands.
- Services are volatile and undergo modifications.

- Services may be competing with each other in terms of their operational similarity.

The objective, therefore, is to provide a seamless on-demand integration of mutually independent services for complex consumer requests. It should be kept in mind that SOA is a set of design principles and does not prescribe a fixed architectural solution model for the underlying problem of service composition. The conventional model for implementing SOA is the 'pull-based' *broker model*. In this model, as will be described in detail in the next chapter, service composition is planned out by a middleware broker and then the composed plan is executed by each participant service by runtime binding over HTTP/communication. Services are invoked as a request-reply process by other services/consumer applications once they get discovered by the broker. This obviously has to be guided by the broker.

Another more recent model is the 'push-based' ED-SOA (Event-driven SOA) that aims to solve some of the problems of broker-based SOA models that will be elaborated in the next chapter. ED-SOA [14] evolved from the EDA (Event-driven Architecture) paradigm and incorporates an *event-manager* middleware (in place of a broker) which observes consumer events (i.e. service subscriptions) and service provider events (i.e. service advertisements) and orchestrates the service composition runtime (instead of computing a composition plan offline) through notifications to both the services and the subscribed consumers. Hence, services are not pulled up but are rather notified by the event-manager about the events that they are "interested" in. In turn whatever output a service generates is again viewed as an event and is notified to all other services/applications that are interested. ED-SOA makes runtime service composition asynchronous and hence, more flexible. ED-SOA provides the core motivation for this dissertation. The dissertation

includes a detailed study of the ED-SOA design paradigm in the context of service composition and investigation of its drawbacks. It then proposes *ALNet* and *SMARTSPACE* as two asynchronous event-driven service composition solution frameworks. *ALNet* looks into the problem from a centralized middleware point of view while *SMARTSPACE* provides a distributed multi-agent based solution platform. The problem of service composition is outlined in the next section.

1.2 Problem Statement

There are at least three components to the problem of service composition:

- Service: Computational entity that is hosted as a service within an SOA-based system.
- Service Description: Advertised machine readable formal schema that describes:
    - Functional parameters in terms of input (I) and output (O)
    - Functional constraints in terms of pre-conditions (P) and effect/result (R)
    - Optional QoS parameters in terms availability, reliability, latency, etc.
    - Service name (textual)
    - Service functionality narrative (textual)
- Consumer query: The query, usually assumed to be in a specific format, given by the consumer to the system. The query must include:
    - The desire of the user
    - Some input information to the system for the services to execute
    - Optional QoS constraints that are desired by the consumer (such as cost or latency)

The general problem then boils down to the following:

Given a dynamic set of services (say *S*) and a consumer query Q find a set of services (say S_c) where $S_C \subseteq S \ni \left(\forall s_i, s_j \in S_C \; ; \; s_i R_C s_j\right) \wedge (S_C \vDash Q) \wedge (f(S_C) \in Min\, f)$

R_C: Order relation that denotes s_i and s_j are composable

f: Cost function (i.e. consumer utility function) that has to be optimized.

From the above problem statement we need to understand that if two services are composable then they are said to form a potential composite service. If this potential composite service satisfies a given query then it is a composite service with respect to that query. It is also worth noting that the set S_C is essentially a partially ordered set over the relation R_C. The query Q is satisfiable (fully or partially) by the set S_C. Hence, S_C is the desired composite service. We intentionally kept the problem statement general because of the following reasons:

- Service descriptions corresponding to the set *S* are defined differently in different approaches.
- Query Q is formally described in different ways in different models and hence, query satisfiability takes on different formal interpretation.
- The relation R_C has been defined in different manner in literature and thus, the underlying notion of composability is also sometimes differently modeled.
- The cost function *f* is modeled differently in different approaches.

Service composition involves two allied problems: (i) *service discovery* and (ii) *service selection*. In order to satisfy a given query the service composer first needs to search

for the necessary individual services that can participate in the composition process. This process is called *service discovery*. However, the composer also needs to optimize the cost function associated with the composition problem. Hence, it needs to select the best services out of the pool of discovered services. This process is called *service selection*. After selection services are then composed into a composite workflow that represents the poset S_C. However, the allied problems of service discovery and service selection cannot be solved as two independent problems separate from service composition. One of the most important reasons for this is that the set of services S is not static for all practical purposes. Services are volatile in nature and may not exist in the same form all the time. Also new services can be added to the system. Another reason is that independent service selection may not lead to a possible S_C due to mutual composability issues later found in the service composition stage. This results in reiterating the selection process again. Hence, we need a unified approach to solve all the three problems dynamically.

1.3 Challenges of Service Composition

The problem of service composition is not easy. The hardness of the problem is because of several factors:

- Services are 'self-described' using a wide range of formal specification languages (such as WSDL [18], BPEL4WS [19], etc.). This creates compatibility issues during service binding since an additional layer is required for inter-format translation. Also the language vocabulary is significantly diverse introducing problems such as semantic

ambiguities due to polysemy and synonymy. This creates lot of problems for service compatibility during composition.

- Huge proliferation of different kinds of services into open systems such as the WWW makes it computationally hard for generating an optimal composition for a given consumer request due to an explosion in the search space.
- Systems are mostly dynamic and hence, any composition process has to take into consideration runtime state changes in the system such as addition of new services and deletion/modification of old ones. This is computationally intensive.
- Services, in principle, are stateless. This means that a service do not store its output state in order to reduce local resource overhead and improve scalability. However, this design principle comes with a cost – service composition during runtime has to be synchronous. The problem with synchronous service composition is that it increases the overall composition latency because of wait periods. This also results in a notification overhead on an external middleware that has to oversee all composition process going on within a system. Moreover, statelessness means that the middleware has to keep track of all the output states and notify services as and when required. This is an additional cost.
- Consumer requests need to be satisfied within a desired time interval and possibly given QoS constraints. This usually requires a trade-off between the optimality of the composition and the global latency of the composition. Also the unpredictability of underlying hardware resources and network resources adds up to the complexity.
- As service composition creates an environment of open pervasive B2B and B2C markets hence, issues such as service reliability, security, safeguarding business policies, legal

validity, validity of transactional properties (i.e. Atomicity, Concurrency, Integrity, Durability), etc. make the composition process very complex.

1.4 Scope & Contributions of the Dissertation

The previous section discussed about the innate hardness of the service composition problem in general. Although all the difficulties discussed therein are very important contemporary research topics over the past decades the dissertation focuses on four distinct (but related) research questions and proposes solutions for each of them. In this section the overall scope and contribution of the dissertation have been outlined.

- **Problem 1 - Service Matchmaking Accuracy & Efficiency**: Formal specifications of service descriptions serve as the building block for service discovery and hence, composition. In a discovery process a consumer query specification is matched with a set of service descriptions so as to retrieve services that can satisfy the query (i.e. services that are *similar* in some sense to the required service stated within the query). This process is called *service matchmaking*. Service matchmaking can be done either at a syntactic level or at a semantic level. In the case of syntactic service matchmaking sophisticated Information Retrieval (IR) techniques are applied to compute specification similarity using vector space similarity measures or information theory based probabilistic measures. However, syntactic approaches suffer from several innate drawbacks that mainly originate from linguistic ambiguity and fuzziness (as discussed in previous section). One such problem is due to polysemy (i.e. case where lexically equivalent terms have different meaning) that leads to false matches during service

discovery. An example of such a problem can be a scenario where a consumer query contains a desire for a service that provides information about the island *Java* while it is incorrectly matched with a service that provides information about the drink *Java*. Another problem is due to synonymy (i.e. case where lexically different terms have same meaning) that leads to false mismatches. An example of synonymy would be a case where the desired service is one that provides information about *Coffee* while it incorrectly misses a service that provides information about the drink *Java*. Other linguistic ambiguities can be purely contextual where a term may have different meanings in different contexts. For an example a query for a service providing *Hot Food* where hotness is the degree of temperature may be incorrectly matched with a service providing *Hot Food* where the meaning of hot is spicy. To eliminate such inaccuracy semantic service matchmaking approaches have been proposed. In semantic service matchmaking the semantics of service descriptions are explicitly formalized into precise logical propositions. A commonly used formalism in most semantic service descriptions is Description Logics (DL) [39]. Description Logics is a predicate logic based calculus that enables terms to be represented as well formed definitions (called *concepts*). Similarity computation is not lexical in these approaches. Instead two concepts are said to be similar if one of the concept's definition satisfies the other concept's definition. Such satisfaction checking, called *subsumption reasoning*, is done using DL reasoners that use techniques such as resolution, unification, tableaux method, etc. However, subsumption reasoning on DL concepts is computationally expensive as the underlying complexity in most versions of DL is intractable (see chapter 3 for further reading). One

of the major contributions of this dissertation is to study this problem in the context of semantic service matchmaking and propose a bit-encoding based solution, called *g-subsumption*, that is non-DL based and performs computation in linear time. More on the approach and the theoretical foundation can be found in chapter 3.

- **Problem 2 - Service Organization Accuracy & Efficiency**: As mentioned in the previous section proliferation of services and the non-deterministic dynamics of SOA based systems leads to the problems of: (i) search space explosion (during discovery) and (ii) maintenance of online registries where service advertisements are stored. A very efficient way to control and prune the search space is to categorically index services in registries. Queries can then be directly mapped on to the category space whose generic description matches with the query description. Most contemporary research works on this problem have applied Machine Learning (ML) techniques such as supervised learning and unsupervised learning of service categories. However, as ML techniques are mostly founded on statistical learning theory therefore the accuracy of such techniques depends on lot of factors that include the underlying parameters of the learning model, the goodness of the dataset in terms of its representative capacity of the entire data space, choice of dimensions (or features), quality of similarity measure and choice of threshold, etc. In this dissertation a non-ML based service category learning algorithm called *Semantic Taxonomic Clustering* (*STC*) has been proposed that utilizes encoded feature set for each semantic service description so as to provide more efficient and accurate learning model in comparison to some of the significant ML based models. *STC* has been covered in chapter 4 of the dissertation.

- **Problem 3 - Computationally Efficient Query Specification**: User query specification is mostly formalized either as *task templates* or as *formal specifications*. In the former approach a query is assumed to be a sequence (linear or non-linear) of desired services and where each desired service (called *sub task*) is formally specified by its input and output. Such a specification requires the consumer to have a detailed understanding of each sub task and the underlying sequence as well. In the case of formal specification based query modeling user queries are represented as a tuple of desired states (both internal and external), required operators, transition function that maps one state to another based on selection of some operators, an initial state, and a set of accepting states that represent the terminating state requirement after the query has been processed. Such a modeling entails that the user should have the knowledge of a specification language as well as the entire state space and operator space. For a lay user both the models are not very useful and practical. Moreover, such models induce a lot of computational overhead since service composition based on task-template is an assignment problem (which is NP-HARD) and service composition based on formal specification is a behavioral equivalency problem (which is also NP-HARD). Since query modeling is intrinsically related to the efficiency of service composition therefore the dissertation also includes a detailed proposal of a novel query model called *Desire-based Query Model* that is shown to be sufficiently expressive to represent simple and complex queries where a user only needs to specify his/her final desired output and initial input without requiring any detailed understanding of the underlying sub tasks or operations and states that might be required to satisfy the given query.

- **Problem 4 - Efficient Design of Event-driven Service Composition**: In the previous section it has been discussed that services in general are stateless. This imposes significant state maintenance overhead on a middleware in an ED-SOA model. This is because under the situation where a particular event channel is busy the middleware has to store all events that are interesting to busy subscribing services/applications and has to forward the stored events by efficiently identifying when the busy channel is ready. Moreover, service composition is still an assignment problem since event notification has to be made to the best subscribing service such that all the selections have mutual interest in each other's event. In the proposed dissertation instead of modeling as an assignment problem service composition has been modeled in two different ways: (i) *non-constrained path optimization problem* (for the proposed middleware based event-driven *ALNet* framework) and (ii) *fair deal game optimization problem* (for the proposed distributed agent-based *SMARTSPACE* framework). It has been observed through theoretical and experimental analysis that service composition efficiency can be significantly improved in both the cases as compared to when composition is modeled as an assignment problem. The principal idea in *path optimization* modeling is to efficiently generate a service network based on the mutual service composability where network nodes represent services and edges represent their inter-dependency. A path optimization problem is: *Given source service node (that can consume part or whole of the given input) and a given end service node (that generates part or whole of the desired output) it is required to find the minimal cost path between the two*. This is essentially the shortest path problem with user defined constraints (such as service quality, service cost,

service latency, etc) imposed as a single utility cost function over the path. When the path optimization takes place in an event-driven framework (that supports proactive participation of the services as opposed to purely reactive participation in conventional ED-SOA) then the problem is called *event-handling*. On the other hand, a *fair deal game optimization problem* is: *Given a set of fair deal maker agents and fair bidder agents where a deal maker agent has a service desire and optimizes its satisfaction utility function while a bidder agent provides a desired service to a deal maker agent and optimizes its corresponding service utility function*. A *fair* agent means that: (i) agents do not manipulate their identity (i.e. role description), (ii) agents cooperate with each other with no bias, and (iii) agents do not behave to obstruct each other's actions. In this model a *deal* is made by a deal maker whenever it has a desire for a set of services.

1.5 Dissertation Outline

The rest of the dissertation begins with chapter 2 that serves as the necessary background for understanding the general problem of service composition along with a detailed summary and analysis of various approaches taken to solve the problem. This chapter is followed by chapter 3 (Semantic Service Matchmaking & Query Modeling) that starts with a brief overview of the dissertation. The chapter then introduces and elaborates on the problem of service matchmaking which is the foundational operation for service discovery and composition. A novel semantic service matchmaking algorithm called *g-subsumption* has been proposed in this chapter that leverages the proposed efficient bit-encoding based *DL-Encoding* algorithm for testing semantic concept subsumption. The chapter then illustrates

how *DL-Encoding* can be utilized to match user queries (represented in the proposed *DQM* format) with services. In chapter 4 (Service Organization by Learning Service Category) we have proposed a novel service category learning algorithm called *Semantic Taxonomical Clustering* (*STC*). *STC* utilizes the *g-subsumption* matchmaking algorithm for clustering service descriptions into their corresponding functionalities. Organizing service description registries into functionally equivalent clusters helps to prune the search space during service discovery and composition. Chapter 3 and chapter 4 set the necessary foundation for introduction of the proposed solution frameworks for event-driven semantic service composition. Chapter 5 (*ALNet* - Event-driven Framework for Service Composition) proposes a novel event-cognition calculus called *Notability Theory* that serves as the formal basis for the proposed event-driven semantic service composition platform called *ALNet* platform. The *ALNet* platform is a centralized framework that facilitates asynchronous service composition by leveraging the underlying service dependency overlay (called *Activity Logic Network*) within an SOA based system. It will be shown that complex service dependency overlays can be simplified into abstract overlays for improving the efficiency of composition. Although the framework is centralized yet it supports pro-active composition of services instead of a notification based reactive model where the entire coordination during service composition is on the middleware. Service composition is formulated as a *event-handling problem* where user requests and service executions are observed and interpreted according to *Notability Theory* by the middleware as well other services as events. Chapter 5 follows up by an alternative approach to the same problem as discussed in chapter 6 (SMARTSPACE: Distributed Multi-Agent based Event Handling) but from a

completely decentralized framework. A JADE agent platform based distributed multi-agent framework called *SMARTSPACE* has been proposed in this chapter. *SMARTSPACE* includes software agents to represent user requests, services, and also act as middle agents for transforming the event-handling problem into an event-driven unbiased game problem where agents deal and bid with each other with the help of minimum local knowledge to win a deal (on behalf of service providers) and to get the best deal (on behalf of users). The dissertation concludes with chapter 7 where it is has been explained and emphasized that the problem of service composition cannot be very accurate under all circumstances if the contextual evidences are overlooked during service composition. The future direction of the research work that has been put into this proposed dissertation is going to be strictly related to the problems of context modeling and context learning especially in a intelligent distributed agent based system as applied to the area of distributed cooperative systems (and specifically in SOA based systems).

All the chapters from chapter 3 to chapter 6 are accompanied by a section of related works that specifically provides a literature review and comparative analysis of the individual problems and techniques that each chapter focuses on. Every chapter also includes a detailed experimental analysis of the performance and accuracy of the various proposed algorithms in terms of system attributes such as domain diversity and size, system complexity and scale, user request complexity.

CHAPTER 2

RESEARCH BACKGROUND

In this chapter we are going to provide the dissertation background and an extensive literature review of research work in the field of SOA based system modeling and specifically in the area of service composition. The chapter first introduces different approaches of modeling SOA. This mainly includes two contrasting models: *broker based SOA* and *Event-driven SOA*. This follows by some literature review of currently existing SOA platforms (both industry and academic). After this the chapter includes an extensive study of formal service description specification languages that serve as the building block for service discovery and composition procedure. This study is followed by a detailed summary of the significant research approaches taken in solving the service composition problem. The chapter concludes with an overview of the FIPA compliant JADE multi-agent platform that has been used in this dissertation as a basis for the proposed *SMARTSPACE* distributed platform for service composition.

2.1 SOA Model

SOA based models can be broadly classified according to two attributes: (i) *initiator* and (ii) *addressee*. From the former perspective the models can be classified into two types: (i) that where communication is initiated by the consumer and (ii) that where communication is initiated by the provider. From the latter perspective models can again be classified into two types: (i) that where communication is direct (i.e. the addressed entity is

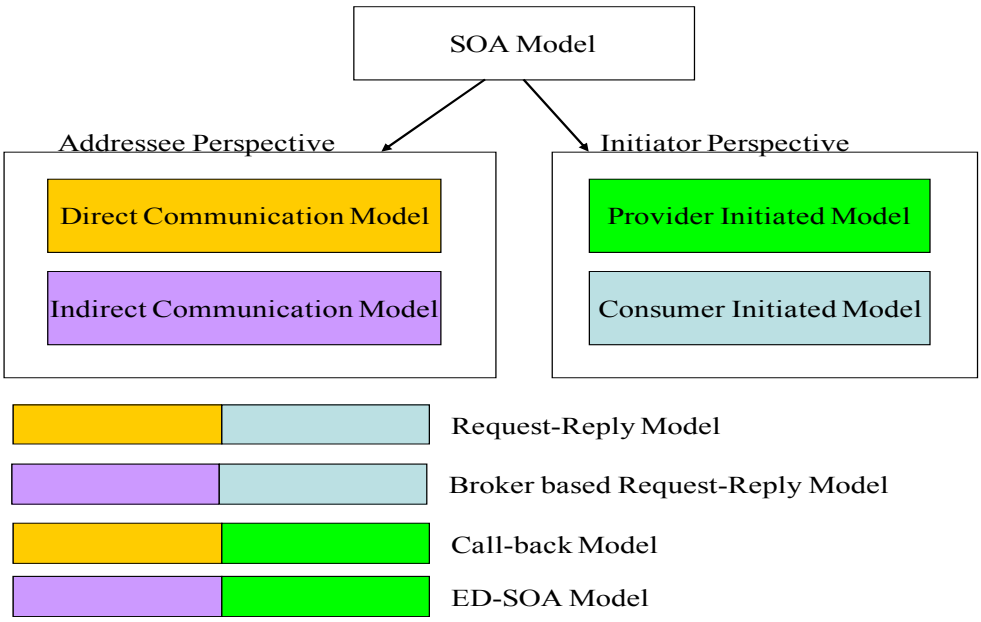


Figure 1: General classification of SOA models

known to the caller) and (ii) that where communication is indirect (i.e. the addressed entity is not known to the caller). In this model of categorization we can place any model into four broad classes: (i) request/reply mode (the consumer is the initiator and directly addresses the provider), (ii) indirect request/reply mode (the consumer is the initiator and indirectly addresses the provider via a broker), (iii) callback mode (the provider is the initiator and directly addresses the consumer), (iv) event-based mode (the provider is the initiator and indirectly addresses the consumer via an ESB). The classification has been summarized in figure 1. In this section we discuss two contrasting modeling approaches: (i) Broker-based SOA models and (ii) Event-driven SOA (ED-SOA) models.

The basic architecture of broker-based SOA model involves three role players: (i) consumer (or user/requestor), (ii) broker (or middleware), and (iii) service provider (figure 2). Service providers typically advertise their service profiles (in terms of formal service

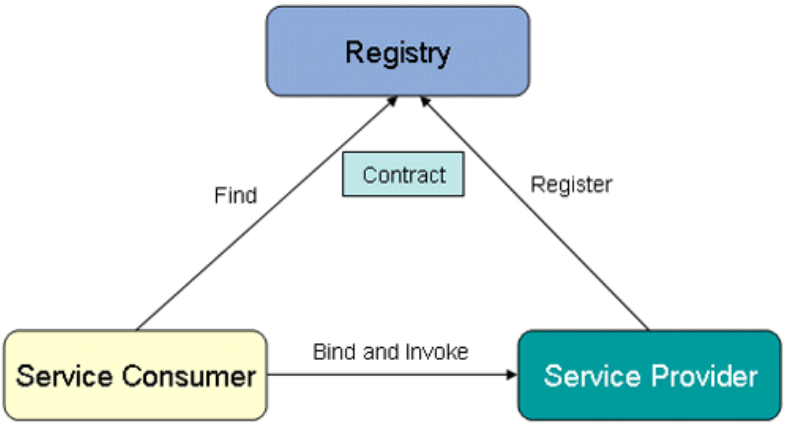


Figure 2: Broker-based SOA Model

descriptions) that are stored and organized in registries maintained by the broker. Registries are organized using formal meta-data and schema represented in languages such as UDDI [8] and ebXML registries [9]. The broker matches the consumer request for a service with the available service profiles and then returns the best matched service profile to the consumer (i.e. applications/services/agents/users). This process is called *service discovery*. The consumer then binds with the corresponding discovered service remotely over synchronous channels (i.e. "*pulled up*") using remote procedure call based protocols such as SOAP [10]. Sometimes the broker has to compose more than one service profiles into a partial order of execution such that the consumer request can be satisfied. This process is called *service composition*. In such a situation the broker supervises the runtime binding of services by providing each of the participating services the service binding information that it needs to for calling the service up. As mentioned in the introduction chapter, the major disadvantage in this mode is that service call-ups have to be synchronous because services are stateless. Synchronous coupling underutilizes resources and introduces integration overhead. Another problem in broker-based model is that the invocation of services needs to

be done explicitly with a priori knowledge of the end-points of such services and possible service contracts. There is no way that the service can do its job and let other services take care of what needs to be done next based on the service's current output state.

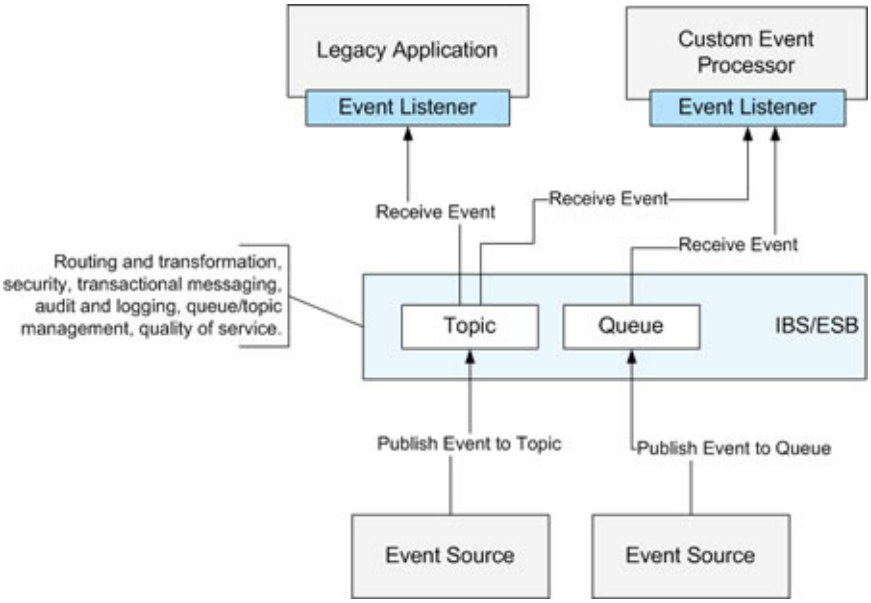


Figure 3: Event-Driven Model

With the advent of Event-driven Architecture (EDA) in software designing a new form of SOA, called *ED-SOA*, has evolved [11 – 17]. Approaches in this paradigm model all processes as events and the communication between these processes as event objects [15]. Processes can be a business decision, a transaction, or any communication signal that may signify an external alarm/interrupt/stimulus to another computing process. Events describe a process completely. An important feature of this paradigm is that it is not assumed that business processes can be pre-designed based on deterministic, static flow of events [14]. ED-SOA model relies on a publish/subscribe mode where consumers and service providers have to subscribe themselves into a registry. A subscription contains the description of the

consumer/provider's interest regarding a particular set of published *event topic*. The ED-SOA middleware maintains the subscription registry along with a well-defined *event library* that stores all possible definitions of notable events in the system. Events are *published* by consumers and providers. In this context, an event can be a business process or a consumer request. Events are pushed into an ESB (Enterprise Service Bus) channel so that they get published into the registry. Event publication is ad hoc in ED-SOA (as in contrast to static predefined UDDI based service advertisements in broker-based SOA). The middleware's role is limited to the mapping of event notifications to consumer subscriptions at the registry. There can be two possibilities when an event is pushed into the ESB: (i) the event is mapped with a consumer subscription and hence, the service is directed (using an *event notification*) to that consumer and (ii) the event is mapped to some other service subscription and hence, the mapped service is triggered (using an *event notification*). Unlike request/reply based SOA models, the event publisher does not need to be aware of the subscribers and vice-versa. In other words, the publisher and the subscriber are fully decoupled. The communication between the processes is asynchronous and hence, event objects can trigger specific target processes without blocking the services from being used for some other purpose. Figure 3 shows an overview of a generic ED-SOA model.

However, there are several drawbacks in ESB based ED-SOA modeling. First, the subscription method entails that subscribers need to know a priori of the event topics at the meta-data level. Hence, publishers need to organize their events into an event taxonomy that may be indexed with unique event IDs. In a dynamic and open system where new types of events are not known a priori such topic-wise subscription cannot take place until the

potential subscribers know about the new topic class. In such a situation event recipients need to be discovered and bound to the events at runtime. This is difficult to do over an ESB-based framework where subscription is independent of event publication. Secondly, classification of events into topics depends upon the way the publishers describe such events. Most descriptions are content based – i.e. they describe the constraints over attributes of the events in an XML format. In this case a particular topic class will contain all events that have the same type of attributive constraints. However, the attributive constraints do not contain information about the event state transition. This restricts several reasoning procedures that require the semantic description of event states. For an example, causality determination, conflict detection, or situation recognition is difficult to reason without properly formalized state descriptions. Thirdly, in traditional ED-SOA models event objects, being concrete data structures, need to be formally created and published by the publishers. As different role players (consumer/provider) may have different (subjective) interpretations of the same event object hence, an event may have several unique IDs that represent different interpretations of the same event. For an example, the event object *flight booked* generated by a flight reservation service may have one interpretation for a hotel reservation service (i.e. a hotel has to be reserved at the destination city for the period of stay) while may not have the same interpretation for a car rental service (i.e. a rental advertisement has to be emailed to the customer for a flight search). In the former case the event is indeed *flight booking* while in the latter case the event is *flight search*. In this case two IDs need to be generated for the same event object. This means that there has to be some mechanism where the consumer interpretations and provider interpretations have to be known and

mapped a priori. This is not possible if the SOA system is dynamic (changes in the system state is frequent) and non-deterministic (the state changes cannot be predicted with certainty) in nature. Fourthly, the data format of the event IDs is mostly at the syntactic level. Hence, such representation makes it difficult to reason whether: (i) a new event belongs to an already known event class and hence, (ii) whether it should be connected to a known set of service classes. In other words, most EDA-based systems lack a proper ontological framework support. Fifth, user request may be difficult to model as events (as understood in traditional ED-SOA systems). The model is essentially a broker based approach where the broker has to satisfy the user request (mostly in the form of a task workflow) by dynamically forming a service workflow that optimally matches the user request. Hence, a complex task-based user request can be seen as a logical workflow of several events that need to be published as a complex event. The published event needs to have subscriptions so that notifications may be sent when the event actually occurs. This is equivalent to saying that the template solution to the complex request event has to be known a priori. In non-deterministic and complex environment such an assumption is over simplification. Sixth, an ESB-based framework may create a bottleneck at the middleware. The load of the middleware can be distributed into a federation of middleware in order to tackle the situation. However, such a federation needs to be updated frequently and the integrity has to be maintained. Also loss of information at the brokers may lead to denial of service. Large federations can also lead to unnecessary network traffic due to forwarding of event notifications and publications mostly in the broadcast mode.

2.2 Service Composition Platforms

Several commercial platforms for service composition have been implemented. Hewlett-Packard's e-speak [20] is an example platform where service descriptions can be registered for dynamic discovery of e-services. Other examples include Microsoft's .NET [21] and BizTalk tools [22], erstwhile Sun Microsystem's Open Network Environment [23], and Oracle Corporation's Dynamic Services Framework [24]. IBM also has its service composition platform called WebSphere Application Server [25]. VerticalNet has come up with a solution platform called OSM [26] that utilizes service ontologies and tools to enhance web service discovery. More academic platforms include that propose an agent-based service matching and invocation [27]. In [28] a service composition platform based on HP workflow has been proposed.

2.3 Service Description Languages

Service description language is the building block for any service composition process. Service advertisements that are published need to be specified formally. In this section we discuss two approaches to service description specification: (i) syntactic and (ii) semantic.

2.3.1 Syntactic Service Description

Several proposals have been made over the past decade on formal specifications for service descriptions. IBM has developed a widely accepted framework for implementing broker based SOA systems and has proposed the three IBM Web service languages: UDDI (Universal Description Discovery & Integration language), WSDL (Web Service

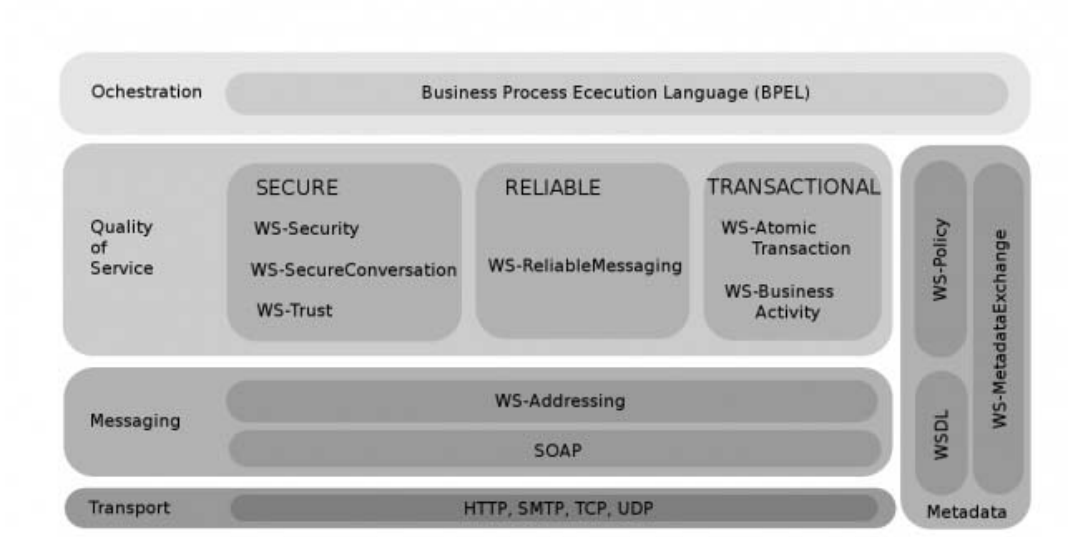


Figure 4: Web Service Standards

Description Language) and SOAP (Simple Object Access Protocol). Each has been briefly described below:

- UDDI [8] provides a registry where service providers can register and publish their services. The registry consists of three parts: white pages, yellow pages and green pages. Contact information, human readable information and related can be registered in the white pages. Keywords that characterize the service are registered in the yellow pages. Service rules and descriptions for application invocations are registered in the green pages (technical).
- WSDL [18] provides the language specification for describing services in terms of: (i) message information, (ii) port information, and (iii) binding information. Services are modeled as *ports*. Port types are abstract collections of operations (i.e. service functions) supported by the service. Messages are abstract descriptions of the exchanged data

between services. Binding constitutes the concrete communication protocol (such as SOAP, HTTP, etc) that need to be followed and the data format so as to call a service.

- SOAP [10] is a proposed W3C standard for exchanging information in a decentralized and distributed environment. SOAP consists of three parts: (i) envelope, (ii) encoding rules, and (iii) Remote Procedure Call (RPC) representation.

All the three IBM languages are syntactic. Hence, linguistic ambiguities such as polysemy and synonymy can rise up very quickly during service matching operation fundamental in all service discovery and composition techniques.

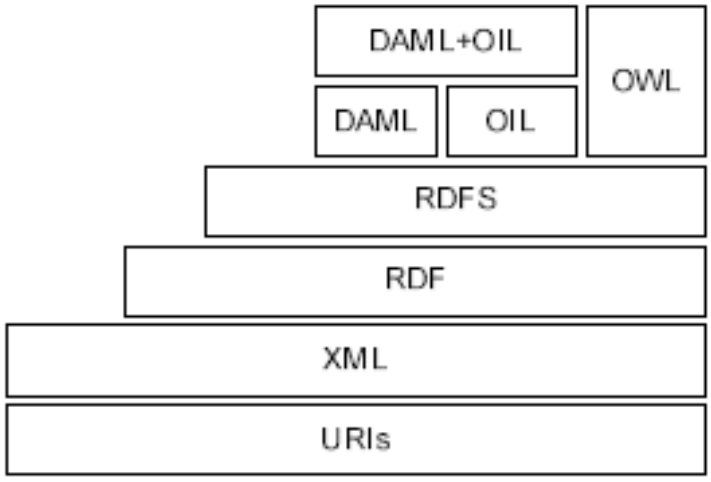


Figure 5: The Semantic Web Standardization Layer

2.3.2 Semantic Web and Semantic Service Description

In order to resolve the problems of linguistic ambiguity service descriptions are recently described using formal languages that have been developed and used in the Semantic Web research community. The Semantic Web is an extension of the current Web in which information is represented as logical well-formed-formulae that can be understood

by computational Web resources and can be shared and processed both by automated tools, such as software agents, and by human users. The vision of the Semantic Web is attributed to Tim Berners-Lee [30]. To understand the utility of Semantic Web we cite the example in [31]: "*Suppose you want to compare the price and choose flower bulbs that grow best in your living area given zip code, or you want to search online catalogs from different manufactures for equivalent replacement parts for a Volvo 740. The raw information that may answer these questions, may indeed be on the Web, but it is not in a machine usable form. You still need a person to discern the meaning of the information and its relevances to your needs*". The Semantic Web addresses this problem in two different ways. First, data is made to be available publicly in machine readable format clearly giving a formal universal meaning to itself. Hence, parsers do not have to parse through the formatting, pictures, ads, and other noises from a web page to get the relevant data. Second, the definitional relationships between different sets of data are explicitly written in machine readable languages. Thus, in the given example a machine can reason that a database with a *zip-code* column has a semantic link with a form that has a *zip* field since the terms (called *concepts*) *zip-code* and *zip* have equivalent definitions. This provides automatic integration of data sources on the Web. In order to ensure common understanding for a particular concept knowledge bases called *ontologies* are used. Ontologies are formal data-structures that store semantic definitions of concepts in a hierarchical structure such that a parent concept definition is satisfied by its child concept definition. Ontologies are used by Semantic Web community as repositories of joint terminology.

The Semantic Web formal specifications have been standardized and recommended by the W3C (figure 5). We briefly discuss each important layer as follows:

- Uniform Resource Identifiers (URIs): It is the foundational specification of the current Web and provides the ability to uniquely identify resources as well as relations among resources. The symbol of a URI includes two parts: an XML namespace and a vocabulary.
- XML (eXtensible Markup Language) [33]: This layer is the fundamental component for syntactical interoperability on Web. XML is the universal format for structured documents and data on the Web. XML-Schemas can be used as an ontology language since it represents the structure, constraints and the semantics of XML documents.
- RDF (The Resource Description Framework) [34]: RDF is a family of XML based languages. It can be used to describe documents in the form of metadata by means of resources (subjects), properties (predicates, describing the resources), and statements (the object, a value assigned to a property in a resource). A description is basically a semantic definition and is called the *RDF triple*.
- RDFS (RDF Schema) [35]: RDFS has been developed as a simple modeling language on top of RDF. RDFS enables the representation of class, property and constraint while RDF allows the representation of instances and facts.
- DAML + OIL: DAML + OIL [38] was built over RDFS to increase the expressivity of the concept definitions (i.e. triples). Although started up as separate projects DAML (DARPA Agent Modeling Language) [36] and OIL (Ontology Inference Layer) [37] were finally converged into a single specification called DAML+OIL by W3C. OIL was

part of the On-To-Knowledge project and was developed as both a representation and information exchange language. The language models primitives from frame-based languages and Description Logic (DL) [39] so as to provide a universal markup language for the Semantic Web. DL serves as a rigorous theoretical foundation for formal logic-based reasoning of semantic definitions.

- OWL (Web Ontology Language): OWL [40] is the latest W3C recommended semantic markup language for publishing and sharing ontologies on the Web. OWL comes in three flavors depending on the expressivity: (i) OWL Lite (Classification hierarchy and simple constraints), (ii) OWL DL (adding class axioms, Boolean combinations of class expression and arbitrary cardinality), and (iii) OWL Full (meta-modeling included).

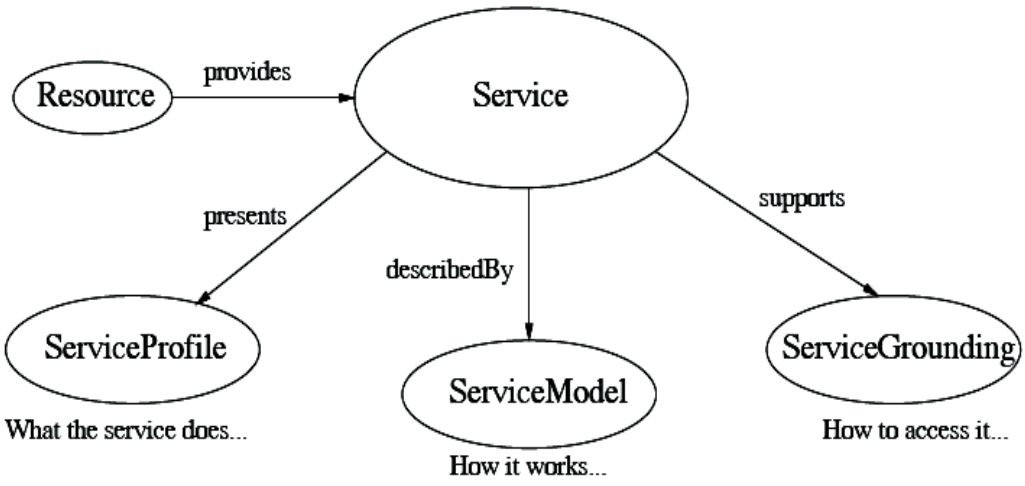


Figure 6: OWL-S Model

DL-based reasoners such as FACT [41], Pellet [42], and RacerPro [43] are used to automatically construct and integrate ontologies. Ontology development and maintenance tools such as OilEd [44] and Protege [45] has been developed and widely used. In the

context of service description there has been significant research. In [46-47] the Web Service Modeling Ontology (WSMO) has been proposed. The objective is to describe relevant features of web services so as to provide a platform for service discovery, selection, composition, mediation, execution, and monitoring. The conceptual foundation of WSMO is Web Service Modeling Framework (WSMF) [48]. WSMO comprises of four root concepts: (i) domain ontologies, (ii) services, (iii) goals, and (iv) mediators. The underlying ontology language is WSML (Web Service Modeling Language) [49] and is not based on any specific ontology language such as OWL. Another web service description language is OWL-S [50-51]. OWL-S is an OWL based service ontology for describing three aspects of web services: (i) Service Profile, (ii) Service Model, and (iii) Service Grounding (figure 6). OWL-S is based on the earlier DAML-S language [52].

Apart from WSML and OWL-S there are other semantic service description specifications that have been proposed. One of them is the Semantic Web Services Framework (SWSF) [53]. SWSF is built on the top of OWL-S and the Process Specification Language (PSL) standardized by ISO 18269 [54]. SWSF comprises of the SWSO ontology that has been written in the SWSL (Semantic Web Service Language) language. Although SWSL is founded on OWL-S yet it is richer and includes PSL as a more expressive procedural model. Another proposal that has its root to the non-semantic WSDL language is WSDL-S [55]. WSDL-S was part of the METEOR-S project [56] discussed in later section. Other WSDL based semantic description languages include SAWSDL (Semantic Annotations for WSDL) [57]. SASDL extends WSDL with a number of attributes that can

be used for the semantic annotation of services. Service operation or XML schema types are annotated with a model Reference (list of URIs) to concepts defined in domain ontologies.

2.4 Static Service Composition

Static service composition problem is tackled mostly in two different ways: (i) Orchestration: calling up each of the services according to a predefined workflow [58] or (ii) Choreography: having each service host execute a predefined set of conversations with other hosts and in the process generate a resultant composite service [59]. Choreographies are written in formal languages such as WS-CDL (Web Service – Choreography Description Language) [60] and UML 2.0 – Sequence Diagram. There has been considerable work on standardization of choreographies between static proprietary orchestrated business processes. RosettaNet [61] is an example in the domain of supply chain. Other examples include SWIFTNet InterAct Realtime for financial services [62] and Health Level Seven (HL7) in the health-care domain [63].

One of the major limitations in this approach is that the user/consumer needs to choose an a priori collaboration (i.e. predefined workflow) of service providers before placing his request. Service composition cannot take place beyond the workflow logic that is provided by the participating vendors. Another major disadvantage is that the services need to be more or less static. We cannot accommodate update of the services in this framework as that might lead to redesigning of part or whole of the service workflow. In a dynamic environment, where services are not stable entities and where service definitions, profiles, internal logic, or service providers may change, this is a serious limitation.

2.5 Dynamic Service Composition

Dynamic service composition, on the other hand, promises to resolve most of the problems mentioned above. However, such a promise comes with a cost – dynamic service composition is not an easy problem. Dynamic service composition approaches can be classified as: (i) *Task-based composition*, (ii) *Goal-based composition*, (iii) *Specification-based composition* and (iv) *Event-driven composition*.

2.5.1 Task based Composition

Many research approaches tend to propose a platform where service composition can take place via a generic middleware that pulls up services from distributed repositories on the basis of a given *task workflow* [64 - 66] where a task workflow is an abstract process model that defines the set of user-given query clauses and their corresponding data dependencies. The query clauses are structured as *desired abstract service templates (also called sub-tasks)*. An example of task-based service composition platform is EFlow [28]. Other research approaches in this direction include [67 - 69].

The objective of task-based composition approaches becomes an assignment problem where concrete services are mapped to the desired abstract services such that a valid and optimal service workflow is established while all the user-specified constraints are satisfied. We call this approach of dynamic service composition as *task-based dynamic composition*. As the assignment problem can also be modeled as a 0/1 Knapsack problem [70] task-based service composition approaches are potentially NP-Hard [71 – 72]. In some task-based models, especially those designed for B2B environments, the task workflow has

to be explicitly specified by the user [73]. However, framing the user-request into a task workflow is not easy as it requires the user to have good understanding of the underlying processes, the data dependency between them, and a process model specification language (like CSDL [73]). Clearly, such a model is not useful for a lay user. Another approach is to let the system frame the task workflow while the user just provides the required sub-tasks and their individual constraints. Even then these models require that the user should be able to frame his request in terms of the underlying processes and their constraints. In several realistic situations, such as making a package tour reservation online or desiring healthcare monitoring services in a smart home environment, task-based models cannot work. Some works have tried to identify generic workflow pattern [74 - 75] for a given query and then match the pattern with existing composite services [76].

2.5.2 Goal based Composition

Apart from task based dynamic service composition, there is an alternative approach for dynamic service composition known as *goal-based dynamic service composition* (or *AI-planning based dynamic service composition*). In this approach the user does not need to define a task workflow. Instead, he/she submits a *goal* that fully defines the required services in terms of their pre and post conditions (or effects) and what they require from the environment in order to give something to the environment [77 – 80]. Services are composed dynamically in such a way to achieve the user's goal considering the pre and post conditions associated with the services. The idea is either to conduct a forward-chain

reasoning where primitive achievable goals are taken up by AI planners [80 – 82], theorem-provers [83 – 84], or rule-based engines [85 – 86] to finally come to the given desired goal.

Research efforts within this approach have been surveyed in detailed in Rao et al. [66]. Some research works have used Situation Calculus based Golog-derivative languages [78 - 79] for specific abstract service procedures and constraints. The approach is agent-based where agents have the reasoning capability to understand user requests and service specifications. There are also works using PDDL (Planning Domain Definition Language) based planners [80]. PDDL based approaches utilize the easy translational property between PDDL and DAML-S. However, such planning is restricted with the closed world assumption and hence, is not suitable for dynamic open SOA-based systems. Another technique for service composition is rule-based planning. In [85] rules of service composability are defined and stored in an expert system. The user query is typically a composite service specification written in CSSL (Composite Service Specification Language). The planner generates a composite plan than matches the user query specification that is the goal in this case. SWORD [86] is an implemented rule based service composition toolkit. SWORD uses the E-R modeling language for specifying services. Service specification consists of the service pre-condition and the post-condition. It is represented as a Horn rule in a pre-condition implies post-condition format. The user query specification is simpler in SWORD and just comprises of the initial state and the final required state. SWORD uses a rule engine for composing the required services to satisfy the query specification. An alternative approach to automated service composition is using Hierarchical Planners such as HTNs. The principle idea is to conduct a backward-chain

reasoning with the help of HTN based planners such as SHOP2 [77, 87] that break up the desired goal into sub-goals until the best set of executable primitive goals are reached. It has been observed that goal-based techniques are computationally expensive as planning involves efficient decomposition of goals into sub-goals, selection of the best set of sub-goals via efficient heuristics, and accurate reasoning of the order in which the primitive goals have to be executed. Moreover, goal-based composition models require that the user should specify the desired goal in terms of one of the possible world states within the domain. In other words, such models are basically closed-world. Also, the plan that is generated is temporal and has to be regenerated for new user goal.

2.5.3 Specification based Composition

Specification based composition (or program synthesis based composition) is another approach to dynamic service composition. The idea originates from the theory of automatic generation of software programs where formal specifications are treated as a theorem and then deductive theorem provers are applied to search for deductive proof of the theorem [88]. The proof is then translated into a program. In the context of service composition a service description is translated into a logical axiom and the required composite service specification is represented as a sequent to be proved. If the underlying theorem prover discovers a proof then a composite service process description is constructed from the proof. In [81] the SNARK theorem prover has been used for automated service composition. Another approach to program synthesis is using intuitionistic propositional logic for service composition [82].

Formal verification of composite service specifications and consistency checking has been a subject of intensive research. Verification is important to check the validity of a composition plan before it can be executed. There are several formal computational model based approaches to model composite service. One such model is Automata (or Labeled Transition System (LTL)) [89 - 91]. In [92] properties of service compositions of BPEL processes communicating via asynchronous XML messages have been analyzed and verified. The underlying model checker is SPIN [93]. SPIN verifies whether service compositions satisfy certain LTL properties. In [94] it has been shown that BPEL/WS-CDL service descriptions can be automatically translated to timed automata and then verified by the model checker UPPAAL [95].

Another approach to composition verification is using Process Algebra based models. Process Algebra provides a strong theoretical foundation of analysis behavioral similarity of two service specifications (called bi-simulation checking). This is useful in formal analysis of service selection where the best service has to be selected out of the set of behaviorally similar services. In Dumas et. al. [96], a model for interface transformation has been proposed. Interface transformation is the method for changing from one behavioral interface to another based on a collection of operators. π-calculus [97] is another specialized process algebra for composition verification. Compositional constructs in π-can be used to compose services in sequential, parallel, and conditional execution order. In Ferrara et. al. [98] CCS [99] has been used to specify and compose services as processes. The Concurrency Workbench 2 is used to validate composition correctness. In [100] BPEL process specifications and the more expressive process algebra LOTOS [101] has been

shown to be inter-translational. Translation helps to deal with temporal property related issues such as compensations and exception handling. The underlying model checker that has been used is CADP [102].

The third popular approach to composition verification is using Petri Nets [103]. Petri Nets provide a formal way of understanding and analyzing concurrent systems. Petri nets are very popular graphical modeling tools in BPM (Business Process Modeling) related fields due to their capability of expressing complex control-flows [104]. In OuYang et. al. [105] a set of mappings from BPEL control-flow constructs to labeled Petri nets has been proposed. Other works in this direction include [106].

2.5.4 Event-driven Composition

Within the ED-SOA paradigm a fairly recent event-service rule based service composition framework has been proposed by Zakir et al. in [107]. In this work events are mapped to set of services via certain ECA-based rules that need to be triggered when the events occur. The composition technique adopted does not check the validity of causality between two services while establishing a path sequence. Instead a forward reasoning is done over a set of tasks (the tasks are computed by a backward reasoning over a set of rules) so as to form a correct sequence of the tasks. However, owing to the dynamic nature of SOA systems the service network grows and shrinks over time. Hence, formulation of such rules may not be feasible in most of the times. Moreover, such rule based framework is closed-world in the sense that actions are defined to be a finite set without any consideration of the possible variants of such actions when the associated contexts change. Thus, a closed-world

rule-based system logically negates all other possible versions of actions that have solution compositions as well. Also, the set of tasks that is deduced from the set of rules represents services that are sufficient for formation of the desired sequence and does not assume any other intermediate service that may be required in order to fulfill such a sequence.

2.5.5 Semantic Service Composition

Semantic service composition is based on semantic matchmaking of service profiles written in languages such as DAML-S and OWL-S. For any semantic matchmaking operation we require formally defined semantic similarity measures. A lot of researchers have focused on this topic. In [108] a similarity measure is proposed for computing the degree of similarity between a service template and an actual service. The measure is based on the syntactic, operational, and semantic similarity. An algorithm is proposed for Web Service discovery using proper interfaces and operational mechanisms for workflow generation [108 - 109]. Service discovering using DAML described Web Services has been proposed in [110] as well. Other works such as [111] have treated the discovery problem from a functionality requirement perspective. Query languages such as Process Query Language (PQL) have been proposed therein to search process models from process ontology. In [112] semantic representations of state, actions, and goals have been proposed as a framework for service composition. [113] proposes a path-optimization based service composition where a sequence of operators (i.e. services) that compute data are connected together using communication connectors. The shortest path is discovered from a search space that consists of the underlying dependency graph between operators. A very

prominent research work in the field of semantic service composition is the METEOR-S project by LSDIS at University of Georgia [114]. The focus of research was to study the use of Semantic Web technologies in the area of service composition and to develop Semantic Web Service and Process specification, semantics-based Web Services discovery, and Process Composition [109, 115]. MWSAF (METEOR-S Web Service Annotation Framework) was proposed as an ontology-driven mark up tool for Web Service descriptions. Translation algorithms were proposed to translate and annotate WSDL files with relevant ontologies [115]. The METEOR-S Web Services Discovery Infrastructure (MWSDI) served as a platform for scalable semantic publication and discovery of Web Services [116]. The research group also proposed the MWSCF (METER-S Web Service Composition Framework) platform that specifies an activity as a semantic activity template. A service ranking function measures the optimality for service selection based on semantic matching and QoS criteria matching [117].

2.6 Reachability Computation in Service Composition

Service composition as a path optimization problem has been studied earlier in [113]. Finding an optimal path as a composition plan has to be preceded by discovering all possible paths between two service nodes in a service workflow. This problem is known as graph reachability problem and has been extensively studied in the field of computational graph theory [118 - 123]. One approach of solving the problem is to traverse the underlying service graph from the vertex u to vertex v using depth first search at run-time (shortest-path). This implementation does not require extra space and the time complexity is $O(m + n$

log n) where m is the number of edges and n is the number of service nodes. But for massively large graphs (i.e. open SOA based system such as the WWW) *m* can be very big. Another approach is to compute the transitive closure matrix of the entire graph. In this way the reachability can be calculate in constant time O(1) although the required storage space is $O(n^2)$.

Table 1: Reachability Algorithms

Schemes	Query Time	Index Time	Index Size
Shortest Path	$O(m + n \log n)$	0	0
Transitive Closure	O(1)	$O(n^3)$	$O(n^2)$
Interval [118]	$O(n)$	$O(n)$	$O(n^2)$
2-Hop [119]	$O(m^{1/2})$	$O(n^4)$	$O(nm^{1/2})$
HOPI [119]	$O(m^{1/2})$	$O(n^3)$	$O(nm^{1/2})$
Dual-I [120]	O(1)	$O(n+m+t^3)$	$O(n+t^2)$
Dual-II [120]	$O(log(t))$	$O(n+m+t^3)$	$O(n+t^3)$

Interval based indexing approach [118] is the best approach for trees. The principle idea is to traverse the tree in pre-order and incrementally assign a lower bound of an interval till the leaf nodes are reached. Then during upward return traversal the upper bounds are assigned incrementally. Two vertices are reachable if the interval of one vertex lies within the interval of other vertex or vice versa. Time complexity for reachability test is O(1).

However, the technique is not suitable for graphs and also for systems that have frequent updates (addition and deletion of service nodes).

The 2-hop labeling scheme [119] is another technique that suits well for sparse graphs. The time complexity of reachability test is $O(m^{1/2})$. However, obtaining the 2-hop labels is NP-hard. Using approximation algorithms the problem was reduced to $O(n^3)$. In the dual labeling scheme [122] the graph is broken down into two components: (i) spanning tree of the graph (ii) and the set of non-tree edges (t). These two components constitute the complete reachability information of the graph. The spanning tree is indexed using the interval based approach and the non-tree edges are kept in a table. At query time the interval based approach is consulted to find out if there is reachability between two nodes. In case an answer is not found then the non-tree edge table. The performance of this approach depends on the value t. The value of t can be reduced by selecting the appropriate spanning tree. This can be achieved by obtaining the minimal equivalent graph. Table 1 summarizes the reachability prominent algorithms.

2.7 Service Composition & Distributed Multi-Agent Platform

Service composition can also be addressed from a multi-agent cooperative model perspective. This is because software agents have adaptive sociability features that help them to collaborate with each other for a common goal. In such a model services are viewed as agent behavior which the agent chooses to adopt based on specific interpretation of the system state and the consumer query. The W3C specifies software agents as "... *running programs that drive web services both to implement them and to access them as*

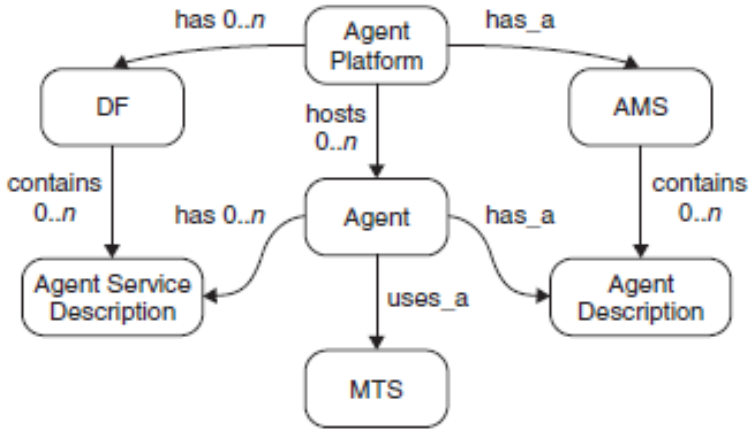


Figure 7: FIPA Agent Management Ontology [125]

computational resources that act on behalf of a person or organization" [123]. Agent-driven SOA models therefore add an additional layer of intelligence, autonomy, and flexibility on top of conventional service composition processes.

Most agent technologies in recent research revolve around the FIPA (Foundation for Intelligent Physical Agents) Agent Management Specifications [124]. According to FIPA any compliant agent platform must have these features: (i) AMS (Agent Management System), (ii) DF (Directory Facilitator), (iii) AC (Agent Container), and (iv) MTS (Message Transport System). We discuss each of them as follows:

- AMS: The AMS is a record keeper agent that creates, maintains, and destroys (i.e. kills) agents in a specific platform governed by it. It assigns unique identifiers to services and keeps a record (essentially a white page) of all active services within a system.
- DF: The DF is an agent that behaves like a yellow page directory service provider in the system. The directory keeps record of all active agent behaviors via the means of

behavior publication (similar to UDDI registry publishing). Discovery of a required agent behavior can be conducted over the DF registry.

- AC: The AC is the runtime environment where an active agent lives. An agent's life cycle management facilities are provided by the AC.
- MTS: The MTS provides the inter-agent communication bus via which agents talk to each other by exchanging speech-act-theory [127 - 128] (standardized into FIPA as FIPA Communicative Act Library Specification) styled ACL (Agent Communication Language) messages. ACL packets are put into SOAP envelope and are typically sent over HTTP, WAP, or CORBA IIOP.

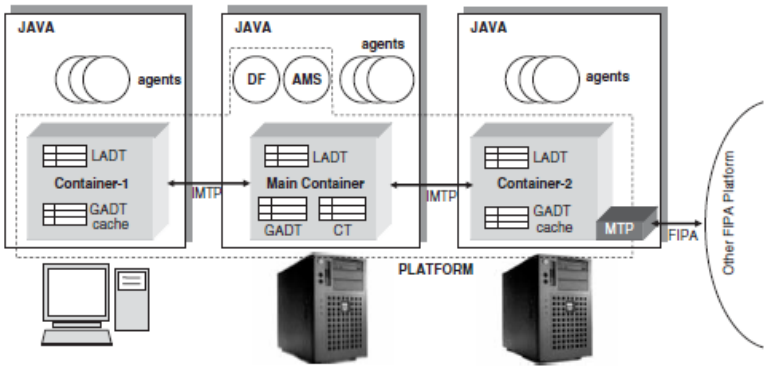


Figure 8: JADE Architectural Overview [125]

Figure 7 depicts the ontological framework of a FIPA compliant platform. There are several research proposals on FIPA compliant multi-agent platforms. Some like the IBM Aglet Toolkit [126] and Telecom Italia's JADE (Java Agent Development Environment) platform [125] have become very popular. In this dissertation we have specifically chosen JADE as the underlying agent development framework for the proposed *SMARTSPACE* service composition platform. JADE is Java based and provides implementation capability

of multiple containers that are linked up with a single main container where the AMS resides (figure 8). Each container maintains two tables: (i) LADT (Local Agent Descriptor Table) and (ii) GADT (Global Agent Descriptor Table). These tables keep track of all agents living locally and globally in the system. Apart from these two tables the main container also maintains an additional table called CT (Container Table) that is essentially the AMS white page. Inter-container communication within the same platform is typically done using IMTP (Internal Message Transport Protocol) that is built over Java RMI.

JADE incorporates the Web Services Integration Gateway (WSIG) for implementing web services as agents. Some very significant work has been done in the area of bridging web service standards and agent technologies [129 - 131]. Gateways are software implementations that bridges web service specifications (WSDL + SOAP + UDDI) to agent technologies (i.e. FIPA) such that web services published in UDDI registries can be accessed by agents and agent behaviors in turn can be published as web services. Gateways typically has a service discovery converter (for inter-operation between UDDI and FIPA DF), service description converter (for translation of description content between WSDL and FIPA SL), and communication protocol converter (for translation between SOAP to ACL).

2.8 Summary

This chapter has given a detailed overview of SOA models and their application in solving the problem of service composition. Extensive literature review has been given in

this area of study. The chapter also includes a summary of FIPA compliant multi-agent platforms and their applicability in the problem of service composition.

CHAPTER 3

SEMANTIC SERVICE MATCHMAKING & QUERY MODELING

3.1 Introduction – The Proposed Framework

Service discovery, selection, and composition are based on formal service descriptions. In general these service descriptions are advertised by providers so that the composer has access to them. We observed that, as mentioned earlier in chapter 1, most formal languages for service description such as WSDL are syntactic. Hence, polysemic and synonymic ambiguities during service discovery and composition induce sub-optimal accuracy. There has been considerable research on solving this problem by adding an extra ontology-driven semantic layer over syntactic service description languages. Semantic languages such as DAML-S [52] and later on the W3C recommended OWL-S [50] evolved in this direction. The principle idea was to replace the token-based vocabulary of syntactic languages by a standardized vocabulary that consists of Description Logics (DL) [39] based *wff* (well-formed-formulae) definitions. This creates the possibility of logical reasoning based computation of service description similarity (called *service matchmaking*) that is the foundational operation for service discovery and composition process. However, DL-reasoner based computation is inefficient and intractable in the worst case. This led to the proposal of a novel encoding based linear time service matchmaking algorithm called *g-subsumption* (introduced in this chapter). The algorithm is based on the proposed dynamic bit-based coding theory, called *DL-Encoding*, that preserves the semantic definitions of service descriptions while reducing the matchmaking operation to simple logical bit-

operations. The proposition of the *DL-Encoding* theory serves as the building block of the rest of the dissertation (figure 9).

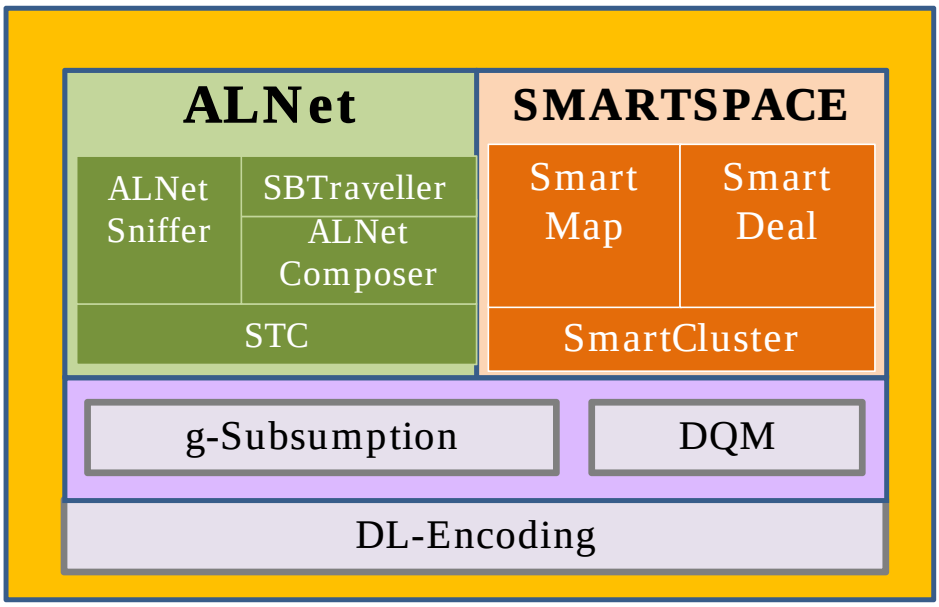


Figure 9: Dissertation Structure

Query specification is another important aspect of service composition. Consumer requests have to be formally specified in a way that is: (i) user friendly with minimum requirement of technical understanding and the hidden process workflow, (ii) comparable with the service description format for easy matchmaking, and (iii) avoids unnecessary complex composition. Conventional *task-template* based query models and *formal specification* based query models do not include these three desirable features. Moreover, they make the compositional problem computationally hard since the problem reduces to an assignment problem [71 - 72]. To support these necessary features we have proposed a new formal query model called *Desire-based Query Model* (*DQM*) that requires the consumer to provide the desired output and whatever input s/he can provide at a given moment. The user

does not need to know the underlying composite process workflow or any formal specification language. Also the formulation is such that the composition problem is no more an assignment problem but rather a constraint-free path optimization problem. *DQM* allows queries to be of three kinds: simple, complex, and compound and hence, is expressive enough to represent a wide range of queries. *DQM* formatted queries are encoded using *DL-Encoding* in the same manner as services are encoded in *g-subsumption*. Together with *g-subsumption* matchmaking algorithm *DQM* forms the basic platform for the proposed solution frameworks of *ALNet* and *SMARTSPACE*. In both these frameworks queries are formatted as per DQM and experimental evaluations involves complex DQM queries only. *DL-Encoding, g-subsumption*, and *DQM* will be the content of this foundational chapter.

For service discovery the composer has to perform a service matchmaking process given a query so as to satisfy the query as stated in section 1.2 of chapter 1. Service discovery can be significantly improved if services can be organized into their corresponding functional categories. This is because categorization can prune the search space for finding a set of services that matches a query specification. More specifically the problem has two parts: (i) modeling a learner that can learn service categories given an unobserved set of services and (ii) mapping the query (in *DQM* format) over the categorized service space to extract services that are similar to the query specification. Lot of research works has studied this problem extensively by applying Machine Learning (ML) techniques. However, we argue that if service descriptions can be mapped into a semantic space then we can model a significantly improved non-ML learner (in terms of accuracy) and at the same time achieve a much better discovery efficiency in terms of computational overhead. This

led to the proposal of a novel service category learning algorithm called *Semantic Taxonomic Clustering* (*STC*) algorithm. *STC* is based on pair-wise *g-subsumption* matchmaking between service descriptions and is an essential component of the *ALNet* framework. *STC* has been proposed and analyzed (in terms of performance and accuracy) in chapter 4 of this dissertation.

The dissertation subsequently unfolds into the proposed *ALNet* framework for event-driven service composition in chapter 5. It has been observed therein that there are some limitations to the existing models of ED-SOA based systems. One of the biggest obstacles is the requirement of an *event library* that incorporates formal definitions of all possible events in the system. Service providers as well as service consumers need to follow a specific event format in order to publish and subscribe to events. However, this is not possible in an open and dynamic system where events cannot be pre-determined. Furthermore, event recognition and event interpretation are coupled together in conventional ED-SOA models. This is a serious limitation for situations where: (i) a single event can have different interpretations for different subscribing applications, (ii) a single event can have different interpretations for the same subscribing application at different time points, (iii) multiple events can have same interpretation to a set of different subscribing applications. Such situations arise when an application such as software agents is intelligent in some sense (i.e. has a belief system and a reasoning capability) and is adaptive to changes in its belief. Thus, the chapter introduces a formal event cognition theory called *Notability Theory* that reconciles such limitations by completely decoupling event recognition from event interpretation. A formal ontological framework, called *CAOFES*, for semantic representation of the theory has been proposed.

The proposed *ALNet* event-driven framework is essentially founded on Notability Theory. In the context of Notability Theory the problem of service composition is called *event-handling*. There are three important components of the *ALNet* framework: (i) middleware service registry organization based on *STC*, (ii) service discovery component called *ALNetSniffer*, and (iii) service composition component (i.e. the event-handling algorithm) called *ALNetComposer*. *ALNet* platform also includes an additional component called *SBTraveller* that optimizes *ALNetComposer* as the platform evolves over time (figure 9).

In chapter 6 an alternate ED-SOA framework called *SMARTSPACE* has been proposed. *SMARTSPACE* is a distributed multi-agent based solution platform to the event-handling problem discussed in chapter 5. In comparison to *ALNet* platform *SMARTSPACE* addresses the issues of large scale high dynamics of SOA based systems where the diversity and the size of services are significantly greater than the scope of *ALNet* and the innate randomness within the systems is also much higher that what *ALNet* addresses. *SMARTSPACE* is built over the Notability Theory as well and models all computational entities as agents and their corresponding services as behaviors. However, *SMARTSPACE* drastically diverges from the conventional notions of SOA, as will be detailed in this dissertation, in many ways. It does not require any centralized middleware composer. All the processes are completely localized and hence, *SMARTSPACE* does not require any global system state knowledge. This makes the framework highly adaptive to the dynamics of the system. The event-handling problem is modeled as a cooperative fair deal game problem by *SMARTSPACE*. Within a game instance agents deal with other agents and in turn also bid to win a deal. Deals finally converge to win-win equilibrium (i.e. confirmation of deals) for

participating agents in an event-handling process. Analogous to *ALNet* platform *SMARTSPACE* also has three components: (i) a service agent organization algorithm called *SmartCluster* that essentially is the distributed version of *STC*, (ii) a distributed service agent discovery algorithm called *SmartMap*, and (iii) a distributed service composition (i.e. deal-bid game based event-handling) algorithm called *SmartDeal* (figure 9).

3.2 Service Matchmaking

One of the key problems in Service Oriented Architecture (SOA) based system is service discovery. The fundamental operation required for service discovery is *matchmaking* of service descriptions against consumer query descriptions. Matchmaking is a computational operation over a pair of service/query descriptions that maps a defined similarity measure function into a real or Boolean space of similarity score. Such matchmaking is also required for clustering service registries into groups of similar services to prune the search query space. Service matchmaking results in significant accuracy enhancement if the service/query descriptions are semantic. Semantic services (e.g., semantic web services) are described using XML-based representational languages such as DAML-S (DARPA Agent Mark-up Language for Services [52]) and more recently OWL-S (Web Ontology Language – Service [50]). The underlying mathematical foundation of most of these languages is Description Logics (DL) [39]. In such a framework service/query descriptions can be modeled as a bag of DL *concepts* that are already defined within a set of domain ontologies. The semantic service matchmaking problem then essentially becomes subsumption testing of *concepts* that have DL based definitions and are used for

semantically describing services [132 - 133]. Most semantic matchmaking algorithms in literature have employed DL-reasoners (such as PELLET [42], FACT++ [41], etc.) for subsumption computation [132 - 134]. However, DL-based subsumption reasoning can be intractable even for relatively simple languages within the DL family [39].

An alternative approach of reducing the subsumption computation significantly is by using encoding techniques to codify ontology concepts [136 - 143]. Such techniques guarantee O(1) time worst-case subsumption computation (considering no in-memory constraint). However, most encoding techniques assume that there must exist a pre-defined base taxonomy of ontological concepts whose mutual subsumption relations are already known. In other words, these techniques have to rely on a DL-reasoner that has to be used to generate the base concept taxonomy. As a result such techniques cannot allow dynamic encoding of service descriptions that uses newly defined concepts not existent a priori within the original taxonomy. In this chapter we propose an alternative non DL-reasoner based linear time (with constrained in-memory) semantic matchmaking algorithm called *g-subsumption*. The algorithm leverages a novel bit-encoding based concept subsumption testing technique. The encoding method converts $\mathcal{ALC}_{\mathcal{R}^c}\mathcal{H}$ (a very expressive DL language) definitions of concepts (used in describing services) into equivalent bit codes preserving the definitional properties of the concepts. Since *g-subsumption* preserves the semantic properties of concepts it is able to support dynamic subsumption testing (unlike most prevalent encoding techniques).

The rest of the chapter begins by identifying and making a comparative analysis of some of the significant existing methods that can be used to some extent for fast concept

subsumption computation. The chapter will then unfold into the details of the proposed encoding theory and its computational feasibility in terms of service encoding. Having laid the necessary theoretical foundation we will then introduce the *g-subsumption* algorithm and make a detailed analysis of empirical studies that support the computational efficiency that we are able to achieve using both standard real datasets (OWLS-TC v2 [134]) as well as on simulated synthetic datasets.

3.3 Semantic Subsumption - Background

As mentioned in the introduction, semantic match making over service description in most cases is DL reasoner based [132 - 133]. Reasoning over DL has been a very extensively studied area. There are three basic kinds of reasoning within the DL framework: (i) *subsumption reasoning*, (ii) *unsatisfiability reasoning*, and (iii) *semantic rule validation*. An informal explanation of each of these 3 types is given:

- *Subsumption Reasoning*: It is a computational operation that checks whether a given concept definition can satisfy (i.e. be a sub-concept) of another given concept definition. For an example, we can check through subsumption reasoning techniques that a concept *car* is a sub-concept of the concept *vehicle* given the DL definitions of both the concepts. In the context of semantic service matchmaking subsumption reasoning can be used to evaluate the three kinds of service matchmaking as described in [132]: (i) *exact*, (ii) *plug-in*, and (iii) *subsume*.
- *Unsatisfiability Reasoning*: It is a computational operation that verifies whether a given concept has at least one set of interpretations that satisfy its definition. For an example,

given the definition of the concept *car* and a set of all possible interpretations we can figure out through unsatisfiability reasoning techniques whether *car* can have member instances (e.g., *Honda Civic Sedan DX*). In the context of semantic service matchmaking unsatisfiability reasoning techniques can verify whether a given service description is semantically well-formed or not. For an example, if a service provider for a car rental service describes the output parameter as $car \cap ship$ then in any current domain the parameter is obviously an impossible parameter.

- *Rule Validation*: It is a computational operation over Horn-like rules (implemented by rule languages such as SWRL [135]) that verifies whether a rule can indeed be satisfied with the current set of interpretations. For an example, a rule such as $\exists x; isA(x, Vehicle) \wedge hasColor(x, RED) \rightarrow isA(x, Car)$ states that all instances of *vehicles* having color *red* must be instances of *car*. Rule validation techniques enable us to verify whether, such as in this example, there can really exist an interpretation *x* that satisfies both the antecedent and the consequence part of the rule. In the context of semantic service matchmaking such techniques can be applied to verify whether a pre-condition rule or an effect expression of a given service description is satisfiable under a given domain or not. However, in the present work since we are not including pre-conditions and effects into our service matchmaking algorithm hence, we will not discuss rule validation within the scope of this dissertation.

It can be shown that all three of these forms of reasoning are inter-reducible [39]. Thus, if we can efficiently solve subsumption reasoning we can also efficiently solve the other two. However, in almost all languages within DL such reasoning has been proven to

be intractable with most results being *NP-hard* and *coNP-hard* [39]. Even in the simplest sub language of DL such as $\mathcal{FL}$ satisfiability is proven to be *coNP-Hard* [144]. These findings imply that a mere increase in the expressivity of DL generates a major increase in the computational cost for subsumption testing.

A way to avoid the aforesaid problem (coined "*computational cliff in DL*") is to generate (and maintain) a taxonomy of concepts where the taxonomy represents a lattice structure over the subsumption relation between concepts within the lattice. Such a lattice structure (also called *inheritance graph*) is formed by pair-wise subsumption reasoning (done by DL reasoners) over a fixed set of concepts. The taxonomy is then encoded and the concept codes are used to test subsumption mostly in constant time (assuming no limit on memory). This way we can avoid expensive re-computation of subsumption over the same concept set. We can summarize such taxonomy encoding into three categories: (a) *bit-vector based* [136 - 140], (b) *interval-based* [141 - 142], and (iii) *prime number based* [143]. We explain each of these 3 approaches as follows:

- *Bit-Vector based Encoding*: A bit-vector is a binary transitive closure table representing pair-wise subsumption (and non-subsumption) with 1 (and 0). There are 3 main approaches that utilize the bit vector: (i) the top-down approach where bit encoding topologically sorts the concept taxonomy starting from the root concept [137 - 138], (ii) the bottom-up approach where encoding starts from the leave concepts [136, 139], and (iii) the conflict-graph based approach where the original taxonomy is reduced to a smaller conflict graph before encoding takes place [138 - 139]. In all these encoding approaches concept subsumption is checked by doing a bit-operation (AND/OR) over

the bit codes. However, in these approaches we cannot avoid the problem of conflicting codes because of a general focus on bit reusability for code compression. Finding and dealing with such conflicts incurs a lot of update costs for generally huge taxonomies. Apart from this problem, for conflict graph based methods, we have an additional computational problem of constructing the conflict graph that actually is based on the *NP-hard* graph-coloring problem.

- *Interval based Encoding*: In this approach [141 - 142] the taxonomy is spanned over using a depth first search assigning incrementally the lower bound of the interval to every new concept visit and then the upper bound of the interval on return upward visits to concepts. However, in cases where the taxonomy is not a tree but a graph where there exist concepts that are sub-concepts of multiple concepts the approach generates additional intervals for all such concepts. Also handling increments (specifically, new leaf concepts) into the taxonomy is problematic. In [142], a linear parameterized function based approach has been used to solve the problem of newer concept additions to the taxonomy. However, the problem of having multiple intervals is still unresolved.
- *Prime Encoding*: This is a top-down approach of encoding a concept taxonomy where prime numbers are assigned to concepts incrementally [143]. Concepts that are subsumed by multiple concepts within the taxonomy inherit the codes of their parent concepts as prime factors along with a new unique prime factor that characterizes the concept. Hence, the final concept code being a product of unique prime factors is guaranteed to be unique. Although this approach clearly resolves the problem of code uniqueness and also multiple codes of a single concept yet there are some significant

drawbacks. One of the major problems is the number explosion of concept codes due to incremental use of primes as multiplication factors. Also, division of large concept code values for checking subsumption is computationally expensive for most computational models and can grow very rapidly as the taxonomy grows.

3.4 Limitations of Taxonomic Encoding techniques

It is to be noted that in all the approaches discussed so far there are some essential assumptions within:

a.) The concept taxonomy has to be generated a priori by a DL reasoner before encoding process starts. This is the primal assumption as any encoding technique requires the taxonomy to exist as the input.

b.) The concept taxonomy has to be static once the service matchmaking process takes off. This means that no new concepts are allowed to be added into the taxonomy after the service matchmaking process is initiated. This is important because if addition of new concepts is allowed then service matching very well reduces to DL reasoner based subsumption testing because to add new concepts a DL reasoner has to be invoked.

c.) Service descriptions must be defined based on a priori encoded taxonomy only. In other words no service provider can describe a service with concepts that are either excluded from the taxonomy or defined dynamically from existing concepts using DL constructs such as union or intersection.

The problems that follow up due to these 3 assumptions (and that which the proposed *g-subsumption* algorithm has been able to solve) are that:

a.) None of the encoding techniques can avoid DL-reasoning and hence, the issue of computational hardness in semantic service matchmaking exists in essence.

b.) All the techniques restrict the service providers to use common concept taxonomy as the base vocabulary for describing services. This severely undermines the inherent capability of the expressive power of semantic service languages such as OWL-S. For an example, if the concept taxonomy has the concepts *car* and *bus* then a car rental service provider cannot express a service that rents both *car* and *bus* since the corresponding concept($car \cup bus$)is not existent and hence, not encoded within the taxonomy.

c.) There is no way to understand whether a service description is a valid description or not from a semantic point of view. This is because all of the discussed encoding techniques assume the correctness of the underlying DL-reasoner that generates the base taxonomy. If the base taxonomy is incorrectly generated then there is no real way of understanding that just by looking into the codes. For an example, if the concept *car* is defined as $car \equiv hasPart.Anchor \wedge hasPart.Wheel$ and a DL-reasoner mistakenly reasons that *car* should be subsumed by the concept *vehicle* then encoding techniques will assign a valid code to *car* even though the concept is unsatisfiable. Moreover, there is no direct mapping of the concept code with the concept definition in any of the discussed encoding techniques.

3.5 Semantic Service Matchmaking

Semantic service matchmaking can be of four types: (i) *exact match*, (ii) *plug-in match*, (iii) *subsume match*, and (iv) *sibling match*. In most research works as in [132 - 134, 145 - 149] the first three types have been included into the service matchmaking algorithms while the fourth type has generally been neglected. Before we can propose the *g-subsumption* match making algorithm we first need to lay down a general background of semantic matchmaking and its 4 cases as follows:

- ***Exact Match***: Exact match is a case of semantic matchmaking between two semantic descriptions (in our context service/query descriptions) where:
    - For every DL concept within one description there is exists a definitional equivalent DL concept within the other description
    - The two descriptions are definitional equivalent.

  For an example, let us consider two services s_1 and s_2 described in DL as follows:

$$s_1 \equiv (hasInput\,.\,CustomerName \sqcap hasInput\,.\,CustomerID) \sqcap (hasOutput\,.\,AutoSpecification \sqcap hasOutput\,.\,RentConfirmation)$$

$$s_2 \equiv (hasInput\,.\,PersonName \sqcap hasInput\,.\,PersonID) \sqcap (hasOutput\,.\,CarDetails \sqcap hasOutput\,.\,RentConfirmation)$$

  In the above example *CustomerName* may be definitional equivalent of *PersonName* in a given domain while *AutoSpecification* is definitional equivalent of *CarDetails* within the same domain. We can also observe that for every DL concept (and corresponding relations such as *hasInput* and *hasOutput*) in s_1 there is an equivalent concept (and relation) in s_2. Thus, this example is a case of *exact match*.

- ***Plug-in Match***: Plug-in match is a case of semantic matchmaking between two semantic descriptions (i.e. service/query description in our context) where:
    - There exists at least one DL concept within one of the descriptions that definitionally satisfies (i.e. subsumed by) at least one DL concept within the other description
    - The former description definitionally satisfies (i.e. subsumed by) the latter.

  For an example, we take two services s_1 and s_3 such that:

$$s_1 \equiv (hasInput\,.\,CustomerName \sqcap hasInput\,.\,CustomerID) \sqcap (hasOutput\,.\,AutoSpecification \sqcap hasOutput\,.\,RentConfirmation)$$

$$s_3 \equiv (hasInput\,.\,CustomerName \sqcap hasInput\,.\,CustomerSocialID) \sqcap (hasOutput\,.\,SedanSpecification \sqcap hasOutput\,.\,RentConfirmation)$$

  In this example, s_3 has a *plug-in match* with s_1 as s_3's input parameter concept *CustomerSocialID* may be subsumed by s_1's input parameter *CustomerID* and likewise, the output parameter *SedanSpecification* of s_3 is subsumed by the output parameter *AutoSpecification* of s_1.

- ***Subsume Match***: Subsume match is just the inverse match of plug-in match. Hence, in the previous example s_1 has a *subsume match* with s_3.
- ***Sibling Match***: Sibling match is a case of semantic matchmaking between two semantic descriptions (i.e. service/query description in our context) where:
    - There exists at least one DL concept within one of the descriptions that definitionally satisfies (i.e. subsumed by) ***OR*** has least common subsuming concept with at least one DL concept within the other description

- There exists at least one DL concept within the latter description that definitionally satisfies (i.e. subsumed by) ***OR*** has least common subsuming concept with at least one DL concept within the former description
- Both the descriptions have a least common subsuming parent description that may or may not exist within the current set of service descriptions.

To illustrate this case we take three services s_4, s_5, and s_6 such that:

$$s_4 \equiv (hasInput\,.\,CustomerName \sqcap hasInput\,.\,CustomerSocialID) \sqcap (hasOutput\,.\,CarSpecification \sqcap hasOutput\,.\,RentConfirmation)$$

$$s_5 \equiv (hasInput\,.\,CustomerName \sqcap hasInput\,.\,CustomerID) \sqcap (hasOutput\,.\,SUVSpecification \sqcap hasOutput\,.\,RentConfirmation)$$

$$s_6 \equiv (hasInput\,.\,CustomerName \sqcap hasInput\,.\,CustomerSocialID) \sqcap (hasOutput\,.\,BusSpecification \sqcap hasOutput\,.\,RentConfirmation)$$

In this example, s_4 has at least one concept (i.e. *CustomerSocialID*) that is subsumed by at least one concept of s_5 (i.e. *CustomerID*) and s_5 likewise has at least one concept (i.e. *SUVSpecification*) that is subsumed by at least one concept of s_4 (i.e. *CarSpecification*). Here we can also note that both s_4 and s_5 have a common subsuming parent description as follows:

$$(hasInput\,.\,CustomerName \sqcap hasInput\,.\,CustomerID) \sqcap (hasOutput\,.\,CarSpecification \sqcap hasOutput\,.\,RentConfirmation)$$

The same goes with s_4 and s_6 except that instead of having concepts that subsume each other s_6 has an output concept *BusSpecification* that has a least common subsuming concept with that of s_4's output concept *CarSpecification*. The least subsuming concept in

this case is *LandVehicleSpecification*. Thus, both s_4 and s_5 can be said to have a common subsuming parent description as follows:

$$(hasInput\,.\,CustomerName \sqcap hasInput\,.\,CustomerSocialID)$$
$$\sqcap (hasOutput\,.\,LandVehicleSpecification \sqcap hasOutput\,.\,RentConfirmation)$$

Here it is to be noted that both the above common subsuming parent service descriptions may be absent from any of the service repositories that contain the service descriptions. In cases where the parent descriptions are absent we term such descriptions as *abstract descriptions*. Later in chapter 5 and 6 we will explain why this case becomes very significant for accurate service matchmaking and discovery.

3.6 *g-subsumption* Service Matchmaking

In this section we propose a novel matchmaking algorithm, called *g-subsumption*, for computing the 4 cases of service matchmaking discussed in the previous section. The algorithm is based upon a new bit encoding technique, called *DL-Encoding*, that dynamically assigns bit codes to service descriptions based on their DL based semantic definitions (as explained earlier). While the detailed analysis of *DL-Encoding* will be discussed in the next section a complete outline of the proposed *g-subsumption* algorithm has been laid out in this section.

3.6.1 Feature Stratification

Feature-stratification is a technique of breaking up a given DL based service description into its corresponding features so as to form several conjunctive sub-descriptions. In general these features would be the four functional features – (i) *Input (I)*,

(ii) *Output (O)*, (iii) *Pre-Condition (P)*, and (iv) *Result (R)*. Each of these 4 functional features is explained below:

- *Input (I)*: *Input* of a service is a sub-description that includes DL concepts that are used to define the types of input parameters of the service. For an example, for the car rental service s_1 the input sub-description is: $hasInput\,.\,CustomerName \sqcap hasInput\,.\,CustomerID$
- *Output (O)*: *Output* of a service is a sub-description that includes DL concepts that are used to define the types of output parameters of the service. For an example, for the car rental service s_1 the output sub-description is: $hasOutput\,.\,AutoSpecification \sqcap hasOutput\,.\,RentConfirmation$
- *Pre-condition (P)*: *Pre-condition* of a service is a sub-description that defines, in Horn-like DL rules, the environment state set required to be satisfied before the service can be invoked. For an example, for s_1 the pre-condition can be $hasPreCondition\,.\,P_{s_1}$ where P_{s_1} is defined as:

  $$P_{s_1} \equiv [Service(s_1)] \wedge [\exists x; DOI(x) \wedge isWeekDay(x)] \wedge$$
  $$\begin{bmatrix} \exists y, z; Customer(y) \wedge hasAge(y, z) \wedge \\ isGreaterThan(z, 18) \end{bmatrix} \rightarrow executable(s_1)$$

  The pre-condition states that if s_1 is an instance of the DL concept *Service* and if *x* is an instance of the DL concept *Day Of Invocation* (*DOI*) such that *x* is a week day and also if *y* is an instance of the DL concept *Customer* such that *y*'s age is greater than 18 then s_1 can be executed.
- *Result (R)*: *Result* of a service is a sub-description that defines, in Horn-like DL rules, the new environment state is generated by the service as a result of its execution. In the

example of the car rental service s_1 the result can be $hasResult.R_{s_1}$ where R_{s_1} is defined as:

$P_{S_1} \equiv [Service(s_1)] \wedge [executed\ (s_1)] \wedge [\exists x; Car(x) \wedge hasOutput(s_1, x)] \rightarrow$

$deductInventory(CarInventory, x)$.

The result states that if s_1 is an instance of the DL concept *Service* and if s_1 is executed and if the output instance of s_1 (x) is an instance of the DL concept *Car* then as effect x is deducted from the DL concept *CarInventory* representing the inventory of cars.

In our context we include only the first two features (i.e. *I* and *O*) while an in-depth study over the other two features is left as a future work. After the given service description is *feature-stratified* each of the two sub-descriptions (i.e. for *I* and *O*) so formed is furthered pre-processed into a data structure called *g-array* where g = (*I, O*). The *g-array* groups all the object concepts within the DL sub-expressions as a set. Hence, the example car rental service s_1 has an *I-array = {CustomerName, CustomerID}* and an *O-array = {AutoSpecification, RentConfirmation}*. After *feature-stratification* process is done each of the *g-arrays* are bit-codified as per the proposed *DL-Encoding* algorithm that will be detailed in the following section. Let us assume, for the sake of the current discussion, that in the above example *I-array* is encoded as *{DLcode(CustomerName), DLcode(CustomerID)} = {M,N}* where *M* and *N* are bit strings and the *O-array* is encoded as *{DLcode(AutoSpecification), DLcode(RentConfirmation)} = {X, Y}* where *X* and *Y* are bit strings. During the encoding phase the *g-subsumption* algorithm does a global *DL-encoding* over the entire *g-array*s by ORing all the individual member *DL-codes* together, thus, forming two corresponding bit string *DL-codes*, say *P* and *Q*. The global DL-code is termed

as *g-code*. Hence, any given DL-based service description is reduced to a *feature-stratified* set of two *g-codes*: {*P, Q*}.

3.6.2 g-subsumption Algorithm

In this section we propose a service matchmaking measure, called *Feature Similarity* (or *FS*), that is based on *feature stratification* discussed in the previous section. *FS* is a relative measure and is defined upon a particular *g-array* for a service description. We define *FS* as follows:

Definition 3.1: *Feature Similarity* (*FS*) is a measure that is defined over the function $\overset{g}{\supseteq}$ (called *g-relation*) that maps a pair of *g-codes* of two services into a 5-ary service match space (denoted *g-M*) = {0, 1, 2, 3, 4} where 1 represents a *sibling match*, 2 represents a *subsume match*, 3 represents a *plug-in match*, 4 represents an *exact match*, and 0 represents no match. ■

The *g-code* sub-space = {1, 2, 3, 4} is called the *space of feature similar g-arrays* (or *g-*M^{FS}). If two given *g-codes* (say, P_1 and P_2 corresponding to the *I-code* of two services s_1 and s_2) can be mapped into *I-*M^{FS} then s_1 is said to *I-feature similar* to s_2 (denoted as $s_1 \overset{I}{\equiv} s_2$).

It is to be understood that *g-relation* is undefined over the four algebraic operations: {+, -, *, /}. However, $\overset{g}{\supseteq}$ generates an order in terms of strength of similarity where the order is defined over the sub-space *g-*M^{FS} as: 4 > 3 > 2 > 1 > 0. The *g-relation* function is implemented within the *g-subsumption* algorithm. In this section we outline the algorithm while in the next section we detail the *g-relation* function (which is essentially the proposed

DL-Encoding technique of subsumption testing). Assigning *DL-codes* to *g-arrays* is done dynamically at the time of hosting of the service by parsing each member concept's DL-definition (i.e. if the concept has not been already defined within the domain ontologies) and then generating an equivalent bit string *DL-code*. More details of the $\overset{g}{\supset}$ function (which is essentially the proposed *DL-Encoding* technique for subsumption testing) are given in the next section.

Algorithm: *g-subsumption*
INPUT: $s_1.gA$, $s_2.gA$ // *g-arrays* of s_1 and s_2
OUTPUT: {0, 1, 2, 3, 4} // the *g-M* space

START
$s_1.g = s_2.g = 0$

/ checkDomain checks whether a given g-array is already defined in the set of domain ontologies */*
If checkDomain($s_1.gA$) == false
    $s_1.g$ = DL-Encode($s_1.gA$) // *dynamically assigns a unique DL-code to* s_1
Else
    $s_1.g$ = getDLCode($s_1.gA$) // *extracts the already existing DL-code for* s_1
If checkDomain($s_2.gA$) == false
    $s_2.g$ = DL-Encode($s_2.gA$) // *dynamically assigns a unique DL-code to* s_2
Else
    $s_2.g$ = getDLCode($s_2.gA$) // *extracts the already existing DL-code for* s_2

FS_strength = $\overset{g}{\supset}$ ($s_1.g$, $s_2.g$)
If *FS_strength* == 1 // *case of sibling match*
    Abstract_Parent = commonLeastSubsumer($s_1.g$, $s_2.g$)
Return *FS_strength*
END

Figure 10: *g-subsumption* Algorithm

3.7 DL-Encoding

In this work we restrict ourselves to a special sub class of DL definitions - $\mathrm{ALC_{R^c}H}$ (nomenclature as per DL norms). DL concepts that are defined using $\mathcal{ALC}_{\mathcal{R}^c}\mathcal{H}$ are significantly expressive in the sense that we can use the basic DL concept constructors $\sqcup, \sqcap, \neg$ (i.e. intersection, union, and negation), role hierarchy (i.e. subsumptive taxonomy of

concept relations), full existential role quantifier (∃) and role value restriction (∀). Note that the language also supports some of its roles (i.e. relations) to be transitive (e.g., $\sqsubseteq, \sqsupseteq, \equiv$) [39]. The objective of this section is to be able to model the *DL-Encode* function that dynamically maps a given $\mathcal{ALC}_{\mathcal{R}^c\mathcal{H}}$ concept.

For a given SOA based system we assume that there exists a terminology Δ^p consisting of *primitive concepts*. *Primitive concepts* are base concepts that cannot be defined any further within a given system domain. However, these concepts may be ordered partially according to the subsumption relation $\sqsubseteq$ using human domain expertise. Hence, a corresponding *primitive concept taxonomy* (T^p) can be formed out of Δ^p. In a similar fashion we also assume the existence of a terminology R^p consisting of *primitive roles* (relations) from which a corresponding *primitive role taxonomy* (T^r) can be formed. Since all DL definitions must be *definitorial* (i.e. should not contain definitional cycles such as *Human* = *loves* . *Human*) hence, all concept definitions can be unfolded into a set of *primitive object concepts* and a chain of *primitive roles* that are used as the predicate chain for defining the (subject) concepts. Thus, the assumption of T^p and T^r is justified.

As has been outlined in the *g-subsumption* algorithm (discussed in the previous section) the encoding can be either static or dynamic. In the context of *g-subsumption* static encoding refers to the case when a concept is already defined within a set of domain ontologies while dynamic encoding is required only when the concept is new and has not been defined apriori. Hence, for static encoding, an apriori defined concept or relation is either *primitive concept* or *primitive relation* or base concept/relation whose subsumptive order with respect to other apriori concepts/relations within the given set of domain

ontologies is already known. Such a set of taxonomies is called the *base space*. Thus, the *base space* includes both T^p and T^r. The proposed *DL-Encoding* algorithm uses this *base space* to dynamically encode concepts that are outside the *base space* but are defined using concepts and relations within the *base space*. For an example, the *Car* and *containsRecord* can be a base space concept and relation respectively. A concept *CarInventory* can exist outside the *base space* that has been defined as $CarInventory \equiv containsRecord.Car$. It is to be noted that *g-subsumption* has the capability of learning new concepts by including concepts that have been encoded into the *base space*.

3.7.1 Base Space Encoding

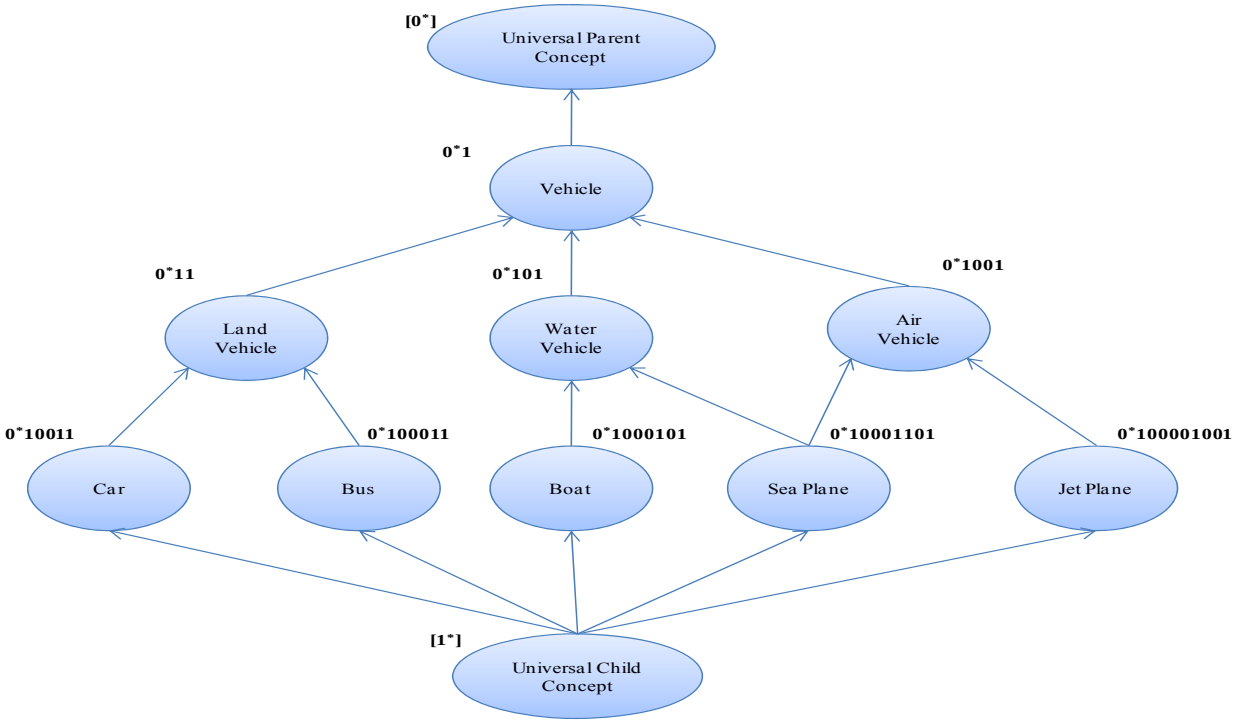


Figure 11: A *Vehicle* Base Ontology (Encoded)

Before we go into the details of dynamically encoding such concepts using *DL-Encoding* we first need to encode the *base space* itself. We first include the universal concept ⊤ and an empty concept ⊥ within a given *base space* to transform the *base space* into a lattice structure where the order relation is the subsumption relation ⊑. The lattice so formed can be seen as a directed acyclic graph, called *base ontology* (denoted as T^{BS}), with a single root concept ⊤ and a single leaf concept ⊥. The root concept is called the *universal parent* (since it subsumes any DL concept) while the leaf concept is called the *universal child* (since it can be subsumed by any DL concept). We define a *parent* and a *child* concept as follows:

Definition 3.2: A *parent concept* c_i is concept within the *base ontology* such that there exists at least one concept c_j within the same *base ontology* such that $c_i \sqsubset c_j$. c_i is said to be the parent of c_j. ■

Definition 3.3: A *child concept* c_i is concept within the *base ontology* such that there exists at least one concept c_j within the same *base ontology* such that $c_i \sqsupset c_j$. c_i is said to be the child of c_j. ■

The *base ontology encoding* algorithm (called *BaseOntoEncoding*) is a simple topological spanning over the corresponding graph starting with assigning code [0^*] to the universal parent ⊤ and finishing with assigning the code [1^*] to the universal child ⊥. The superscript [*] means that the 0/1 is repeated over an *n* bit string length where *n* refers to the current number of concepts within the *base space*. During the topological spanning a new 1 bit, called the *most significant bit*, is assigned to a concept that signifies its unique identity. The assignment is done at the code string position that corresponds to the visit count (or

order) of the topological span. For an example, in figure 11, the order of visit to the concept *Car* is 5. Hence, a 1 bit is assigned to the 5th position of the code string. While assigning code to a concept at a particular visit all the codes of its parent concepts are ORed together so that all of their respective uniqueness can be inherited. This ORed code is then concatenated together with the newly assigned 1 bit. Thus, the concept *Car* in figure 11 is encoded as 0^*10011 where the 1 bit at position 1 is inherited from the code of its sole parent *LandVehicle* (whose code is 0^*11). Codes assigned in this manner are called *b-code*s. The algorithm is outlined below:

Algorithm: ***BaseOntoEncoding***
Input: $\mathcal{T}^{BS}$
Output: $e\text{-}\mathcal{T}^{BS}$

START
Do
  topologicalSort&Enqueue(Q, $\mathcal{T}^{BS}$)
Q[0].code := [0*] // *Q[0] is universal concept* ⊤ */
For each Q[i] s.t. i = 1 to N // *N is the base ontology size*
  Q[i].code := $\bigvee_1(j = 1)$[[Q[i].parent[j].code]] $\vee 2^i$
END

Figure 12: BaseOntoEncoding Algorithm

A very important property of the above encoding algorithm is that it always assigns a unique code to any given concept within the *base ontology*. The uniqueness is guaranteed by the assignment of the most significant bit during the topological span.

3.7.2 Base Concept Subsumption

Once the *base ontology* T^{BS} is encoded as discussed in the previous sub-section we can very efficiently compute whether two given base concepts are mutually subsumptive by using the following theorem:

Theorem 3.1 (***Base Subsumption Testing***): $c_x \sqsubseteq c_y$ iff $c_x \sqsubseteq c_y \rightarrow \left[b-code(c_x) \vee b-code(c_y) = b-code(c_x)\right]$ where $c_x \wedge c_y \in T^{BS}$.

Proof*:* If $c_x \sqsubseteq c_y$ then c_x inherits all the 1 bits of c_y according to *BOEncoding* algorithm (figure 12). Hence, $\left[b-code(c_x) \vee b-code(c_y)\right]$ contains all the common inherited 1 bits of c_y and c_x. For the non-inherited 1 bits of c_x there can only be corresponding 0-bits of c_y since all the 1-bits of c_y has already been ORed up in $\left[b-code(c_x) \vee b-code(c_y)\right]$. Thus, $\left[b-code(c_x) \vee b-code(c_y)\right]$ will also contain all the non-inherited 1 bits of c_x. Therefore, $c_x \sqsubseteq c_y \rightarrow \left[b-code(c_x) \vee b-code(c_y) = b-code(c_x)\right]$.

If $\left[b-code(c_x) \vee b-code(c_y) = b-code(c_x)\right]$ then all the 1 bits of c_x is preserved in the result. Now if c_y is not identical with c_x (i.e. $b-code(c_y) \neq b-code(c_x)$) then there can be two cases: (i) *b-code(c_x)* must contain a set of 1 bits that are not contained in *b-code(c_y)*, and (ii) *b-code(c_y)* must contain a set of 1 bits that are not contained in *b-code(c_x)*. The second case is a contradiction to the initial assumption that $\left[b-code(c_x) \vee b-code(c_y) = b-code(c_x)\right]$ since the result after ORing will be *b-code(c_y)* instead of *b-code*(c_x). Hence, the first case is true. Since all the 1 bits of *b-code(c_y)* are common to *b-code(c_x)* therefore it implies that $c_x \sqsubseteq c_y$ ■

The above theorem proves that the *g-relation* function $\overset{g}{\supset}$ is sound and complete over the base ontology T^{BS}. The time complexity of *g-relation* over T^{BS} is $\Theta(N/W)$ where N is the total word length of a particular computational model (e.g., 64 bits for a fairly modern computer). A comparison of number of base space members (concepts and relations) while W is the word length of the memory for *g-relation* with other DL-based reasoners for subsumption computation has been discussed in the results section.

An example for *b-code subsumption* can be that of the concept *Car* and the concept *LandVehicle* in figure 11. $Car \sqsubseteq LandVehicle = 0^*10011 \vee 0^*111 = 0^*10011 = Car$. Thus, the concept *LandVehicle* subsumes the concept *Car*. In the next section we will introduce the theoretical foundations of DL-Encoding and *g-subsumption* over concepts that are dynamically defined outside the *base ontology*.

3.7.3 DL Bits for Semantic Equivalency

Base ontology encoding is based on the assumption that at least the primitive terminologies (T^p and T^R) exist as a standard agreement within the system domain community. However, *DL encoding* can be applied to the $\mathcal{ALC}_{\mathcal{R}^c\mathcal{H}}$ space outside the *base ontology*. Any $\mathcal{ALC}_{\mathcal{R}^c\mathcal{H}}$ concept definition into 5 simple semantics:

Semantics 1. $C_i = C_j \sqcup C_k$ where $\sqcup$ represents concept union.

Semantics 2. $C_i = \neg\, C_j$ where $\neg$ represents concept negation.

Semantics 3. $C_i = C_j \sqcap C_k$ where $\sqcap$ represents concept intersection.

Semantics 4. $C_i = \forall\, R\, .\, C_j$ where $\forall$ represents value restriction over role R.

Semantics 5. $C_i = \exists\, R\, .\, C_j$ where $\exists$ represents full existential restriction over role R.

Based on such semantic possibility we discuss each of these 5 semantics and their corresponding *DL-Encoding* rules in this section. In order to assign a semantically equivalent code for each of these 5 possible semantics *DL encoding* uses 5 types of bits: (i) *1-bit* used for representing the presence of certain semantic characteristics of a concept, (ii) *0-bit* used for the absence of certain semantic characteristics of a concept, (iii) *X-bit* (called *block bit*) for blocking a position in the code string of a concept to distinguish it from other concepts (use of this bit will be discussed later), (iv) *C-bit* (called *confusion bit*) that is used when it cannot be told for sure whether the bit is a 1 or a 0, and (v) 0^* *bit* for representing the upper end 0s (called *prefix 0*) of a *DLcode* (figure 11). We now define the 0^{th} rule of DL-Encoding as follows: **Rule 0** (**Code Expansion**): If a particular $DLcode = 0^*P$ where P is a code string then it can be equivalently coded as 0^*X^*P where X^* is the *block bit* string.

According to rule 0 if, for an example, the code of the concept *Vehicle* is 0^*1 then as per need it can be expanded to $0^*X1 = 0^*XX1 = 0^*XXX1$ … and so on. Codes are expanded when two codes are treated as operands of any particular DL (binary) operator.

3.7.4 DL-Encoding of Union

An example of semantics 1 in the previous section can be a car rental service whose *O-array* is defined as $\{Car \sqcup Bus\}$ (figure 11). The *O-array* is outside the *base ontology* and hence, has not been defined although individually the concepts *Car* and *Bus* are defined within the *base ontology*. The rule for dynamically generating the new code is:

Rule 1a (**Union Encoding**): $(A = B \sqcup C) \rightarrow [DLcode(A) := (DLcode(B) \ \overline{\wedge} \ DLcode(C))]$ where $\overline{\wedge}$, called *DL-union*, is defined as per the given truth table (table 2).

In the example above the corresponding *DLcode* of the *O-array* is 0^*C01. This code signifies that the first 1-bit is a common bit to both *car* and *bus* while the next 0-bit is a mismatch. The third C-bit distinguishes the union from other possible unions (such as $Car \sqcup Jeep$ = 0^*C001 and Car ⊔ Bus ⊔ Jeep = 0^*CC01). It has to be understood that $DLcode(Car \sqcup Bus) \neq DLcode(Vehicle)$ according to figure 11 since *DLcode(Vehicle)* when expanded becomes *0*XX1*. As mentioned in rule 0, the *X-bits* help to distinguish *Vehicle* with the concept (Car ⊔ Bus). This distinction is necessary to understand that the union operator does not imply any cover axiom over here where we can say that $Vehicle \equiv Car \sqcup Bus$. The concept *Vehicle*, as per figure 11, also subsumes the concepts *Jeep* and *Bicycle*. The *confusion bit* (*C-bit*) may appear because of negation operation done over any concept as will be discussed in the next section. Since the C-bit denotes no proper understanding of whether the bit is 0 or 1 hence the union operator always outputs a C-bit as a result.

Rule 1b (**Cover Union Encoding**): $(A = B \sqcup C) \rightarrow [DLcode(A) := (DLcode(B) \wedge DLcode(C))]$

In case of the presence cover axiom we simply perform a logical AND. Note that *rule 0* does not apply in this case as we do not need the blocking bit to distinguish concepts. Thus, DLcode(Car ⊔ Bus) = 0^*1.

Table 2: Truth Table for DL Union Operator

$\overline{\wedge}$		
1	*1*	*1*
1	*0*	*0*
0	*1*	*0*
0	*0*	*0*
X	*0*	*0*
0	*X*	*0*
X	*1*	*C*
1	*X*	*C*
X	*X*	*X*
C	*1*	*C*
1	*C*	*C*
C	*0*	*C*
0	*C*	*C*
C	*C*	*C*

3.7.5 DL-Encoding of Negation

A concept *A* can be defined in terms of the negation of already encoded concept *B*. For an example, in the case of a car rental service description we can have the *O-array* defined as $\{\neg\, Car \sqcap \ldots\}$. This output is interpreted as any concept that is not equivalent to the concept *Car* but something else. The rule for encoding negation is as follows:

Rule 2 (**Negation Encoding**): $(A = \neg\, B) \rightarrow DLcode(A) := \neg\!\!\!=\, DLcode(B)$ where $\neg\!\!\!=$, called *DL-negation*, denotes a special NOT operator and has the given truth table (table 3).

The C-bit denotes that in the case where a bit is 1 a negation may or may not be flipped into a 0 bit. The only exception is that of the *significant bit* (denoted as 0^*1). Since the *significant bit* characterizes a concept hence, that characteristic has to be removed from its *DLcode*.

Table 3: Truth Table for DL Negation Operator

$\neg\!\!\!=$	
1	C
0^*1	0
0	C
C	C
0^*0 (primitive)	$\perp$ (1^*1)
0^*0 (non-primitive)	0^*1
$\top$ (0^*)	$\perp$ (1^*1)
$\perp$ (1^*1)	$\top$ (0^*)

However, other inherited characteristics (non-significant 1-bits) may not be necessarily removed for a negation operation. For an example: $\neg\, Car$ may actually mean *Bus* where both share the first 1-bit. On the other hand $\neg\, Car$ may also mean something non-vehicle such as *Table* that shares no 1-bit with *Car*. In fact, *Table* can actually have some 1- bits that are not present in *Car*. In other words, some of the 0-bits of *Car* can actually be flipped into 1-bits. However, this is not necessarily true always. Hence, the *DL negation* of *Car* is *0*0C*.

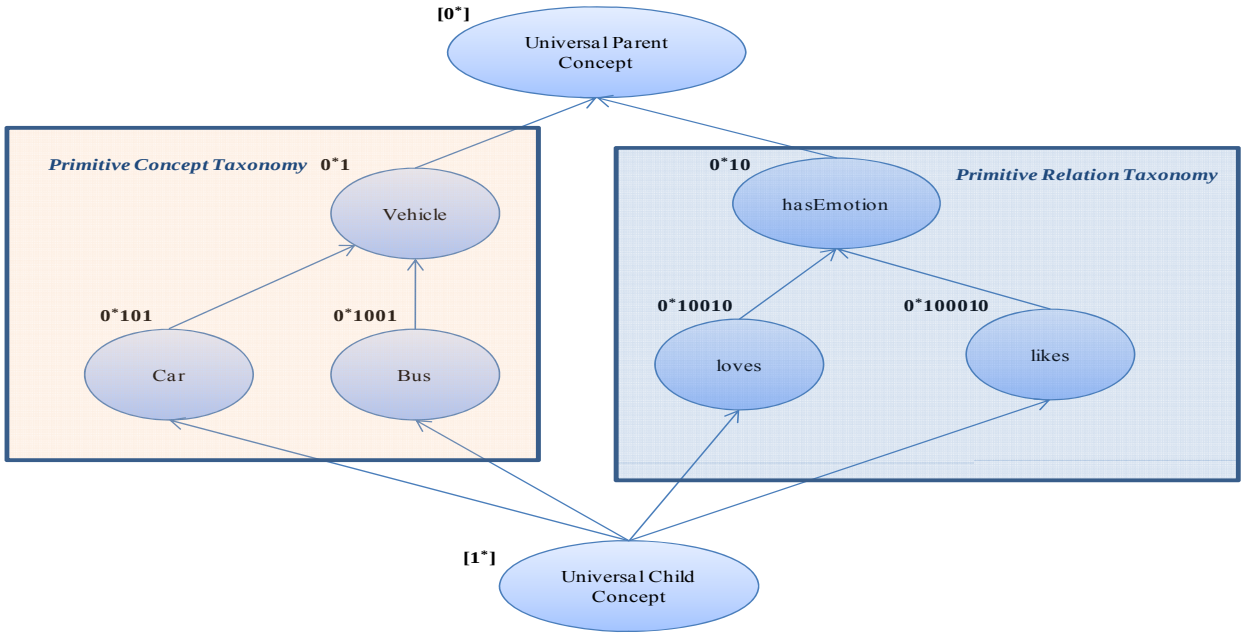


Figure 13: *Base Ontology* with Dual Taxonomies (encoded)

3.7.6 DL-Encoding of Intersection

A concept can also be defined as an intersection of two concepts. DL-Encoding follows de Morgan's law in this case and hence, the rule for concept intersection can be formulated using rule 1 for union and rule 2 negation as follows:

Rule 3 (**Intersection Encoding**): $(A = B \sqcap C) \rightarrow \begin{bmatrix} DLcode(A) := (DLode(B) \bar{\vee} Dlcode(C)) \\ := \neg(\neg DLcode(B) \bar{\wedge} \neg DLcode(C)) \end{bmatrix}$.

Thus, for the car rental service example, if the *O-array* is defined as $\{Car \sqcap Bus\}$ then the corresponding *DLcode* is $\neg(0^*0CC)$ as per figure 13. According to the *DL-negation* truth table the code comes to $1^*1 = \perp$. Thus, *DL-Encoding* is able to show that the concept $\text{Car} \sqcap \text{Bus}$ is unsatisfiable (or invalid). This is so because there is cannot exist a concept that can inherit *all* the properties of *Car* and *Bus* such that it can represent an entity that is both *Car* and *Bus* at the same time. It is interesting to note that there can be no valid semantics 3 concept within the *primitive concept taxonomy* T^p. This is so because of the innate mutual disjointness of *primitive concepts* as per definition. A semantic 3 concept can only be valid if the concept is not primitive. In that case the *DLcode* of $\text{Car} \sqcap \text{Bus}$ would have been 0^*1CC.

3.7.7 DL-Encoding of Value Restriction

In semantics 4 a value restriction restricts the filler concept of the definition of a concept (say *A*) to a particular concept only (say *C*) via the role *R*. Hence, all the 1-bits of *R* and the 1-bits of *C* must be contained within the *DLcode* of *A*. The rule for value restriction uses the normal OR operator and is as follows:

Rule 4 (**Value Restricted Encoding**): $(A = \forall R\ .\ C) \rightarrow [A.code := (DLcode(R) \vee DLcode(C))]$

For an example we can think of an *O-array* = {*CarLover*} where $CarLover = \forall\ loves\ .Car$. This is a very restrictive definition where a car lover can only love car and nothing else. In this case the corresponding *DLcode* as per figure 13 will be 0^*10111. If we

consider another *O-array* = {*BusLover*} where $BusLover = \forall\, loves\,.\,Bus$ then the corresponding *DLcode* will be $0^{*}11011$.

Table 4: Truth Table for DL existential Operator

$\overset{\exists}{\vee}$		
1	*1*	*1*
1	*0*	*1*
0	*1*	*1*
0	*0*	*C*
X	*0*	*C*
0	*X*	*C*
X	*1*	*1*
1	*X*	*1*
X	*X*	*C*
C	*1*	*C*
1	*C*	*C*
C	*0*	*C*
0	*C*	*C*
C	*C*	*C*

Table 5: Truth Table of DL OR Operator

$\overset{o}{\vee}$		
1	*1*	*1*
1	*0*	*1*
0	*1*	*1*
0	*0*	*0*
X	*0*	*C*
0	*X*	*C*
X	*1*	*C*
1	*X*	*C*
X	*X*	*C*
C	*1*	*C*
1	*C*	*C*
C	*0*	*C*
0	*C*	*C*
C	*C*	*C*

3.7.8 DL-Encoding of Full Restriction

Rule 5 (full-restricted role encoding): $(A = \exists R.C) \rightarrow DLcode(A) \coloneqq DLcode(R) \overset{\exists}{\vee} DLcode(C)$ where $\overset{\exists}{\vee}$, called the *DL-full existential operator*, has the given truth table.

In the above truth table the C-bits denote that any of the two binary possibilities can happen. Thus, for the car rental service example, if the *O-array* = {*CarLover*} where $CarLover = \exists loves.Car$ then the corresponding *DLcode* will be 0^*1C111 as compared to 0^*10111 (encoding of *CarLover* in the previous section). Here we see that the encoding has all the necessary 1-bits of the concept *Car* and the relation *loves*. However, it is not restrictive on what they lack (i.e. the 0-bit) and hence inserts a C-bit that can be either a 0 or a 1.

3.8 Dynamic Concept Subsumption

$\mathrm{ALC_{R^c}H}$ concept subsumption testing for dynamic concepts (i.e. concepts defined outside the *base ontology*) is essentially similar to that of *base concept subsumption* testing (as shown earlier). However, instead of the ordinary logical OR operator as was used earlier a modified OR operator, called *DL-OR* (denoted as $\overset{O}{\vee}$), is used for subsumption testing. The DL-OR opertor follows the given truth table.

In the example of the two concepts CarLover and BusLover that we introduced in the previous section we can now try to understand their mutual subsumptive relation in terms of g-relation. Both the concepts are defined outside the base ontology T^{BS} and hence, we cannot use the standard base concept subsumption in this case. Thus, according to the DL-OR operator, the subsumption test is:

$\left[CarLover \overset{O}{\bigvee} BusLover\right] = \left[0^x10111 \overset{O}{\bigvee} 0^x11011\right] = 0^x11111$. The test confirms that the two concepts do not have any mutual subsumption (no exact/plug-in/subsume match) since none of the concepts came out as the final result.

Time Complexity Analysis: There are 10 cases in the truth-table that are applicable for non base concepts (the first 4 are basically equivalent to simple logical OR). If the word length for a particular computational model is assumed to be W then for any pair-wise subsumption testing we are basically dealing with (N/W) chunks of W bits per operand where N is the total number of bits in the DLcode. Thus, in the worst case we have 10 * W cases to check per chunk during subsumption. Thus, the worst case time complexity of DL-OR is O(10 * W * N/W) = O(N).

However, the test does not guarantee that the two concepts are mutually disjoint (i.e. they do not have a sibling match). To verify that we just do a DL-union operation over the two concepts to get the least common subsuming concept (i.e. abstract parent). The abstract parent DLcode for this example is 0^*10011. The way to interpret this code is to begin with the beginning right hand most bit (which is a 1-bit) and check its position. In this case it is the 5th bit. We then go to the 5th topologically ordered concept/relation in the base ontology. In this case it is the relation loves. After that we move to the next 1-bit and check whether that bit is present in loves. The next 1-bit is in the 2nd position and is contained in loves. So the interpretation is still the relation love. Then we move on to the next 1-bit which is the 1st bit. This bit is not present in loves. So we again look up the 1st topologically sorted concept/relation in the base ontology. We observe that it is the concept Vehicle. Thus, the

abstract parent is a labeled concept that is defined: $loves.Vehicle$. If the *abstract parent concept* is the *universal concept* $\top$ then the concepts are disjoint. Also sometimes the *abstract parent concept* is too abstract to be defined. An example of that can be $CarLover \sqcup Car$. In this case the *DLcode* of the *abstract parent* is 0^{*}C0101. Since the beginning bit is a C-bit it is not possible to label the abstract parent semantically as it was for $CarLover \sqcup BusLover$. However, we can observe that there is a semantic relatedness between the concepts *CarLover* and *Car* since there are some 1-bits present in the *abstract parent*. Semantic relatedness is outside the scope of this current work.

3.9 Query Modeling - Background

Most of the previous works have focused on the former where a query is structured as a *task-template* before it is submitted to the service discovery and composition system [64 - 65]. Service composition engines reformat user-requests into *abstract task templates*. Such well-defined task templates are framed as a graph or workflow of *abstract services* (i.e. the required services). These abstract services are the sub goals that constitute the global goal (i.e. the desire of the requestor) to be achieved in the best way through a service composition process. In the case of modeling query as *specifications* the user is assumed to know how to express his/her query in a specification language (such as petri net [104 - 106], FSM [81 - 95], etc) that formally states the given input, required output, and all internal process (and intermediate states) that transforms the given input state to the required output state. However, there are some serious disadvantages in each of these two popular modeling

techniques. We first discuss the disadvantages of the most popular query model – *task templates*:

(a) ***Problem of task structuring***: A task-based approach demands the requestor or query analyzer to format the query into a sequence of sub tasks and model them as an abstract DAG of desired services (termed *abstract service*). This assumes prior knowledge of the type of services that may be required to satisfy the user request. In other words, it demands a thorough design process of the user request at the first hand. This again is a costly overhead. At the same time the temporal and functional dependency order in which the desired services have to be invoked should also be specified by the requestor or the query analyzer. This assumption happens to be very weak when the requestor is a lay user (as depicted in our use case in section 3.10).

(b) ***Problem of task analysis***: Query processors have to analyze a formally represented query task and map that into an isomorphic sub graph of existing services from the underlying service dependency graph. This is basically a service assignment problem. Numerous research efforts have modeled this problem of service composition with multidimensional QoS constraints as a mixed integer programming problem. It has been proven that this problem can be transformed to the Multiple Choice Multiple Dimension Knapsack problem and hence is NP-Hard [71 - 72].

(c) ***Problem of task mapping***: There may be many situations where the query is not a one to one mapping with the services available. This is especially so when there can be services that can satisfy a particular sub task partially while the rest of the sub task can be satisfied by some other service.

(d) ***Problem of task recognition***: Queries may not evidently be recognized as a task. There may be a lot of cases where user interactions have to be interpreted as events rather than requests and subsequent services have to be invoked on the fly to handle those events. For an example, a person having a heart attack may have to be monitored and interpreted as an event (rather than a specifically designed task template) and corresponding healthcare services have to be invoked so that the person gets the proper hospital treatment.

(e) ***Problem of selection incompatibility***: In most task-based approaches services are selected independently before the service composition takes place. However, the best selections may not be functionally compatible with each other during composition. In this context compatibility means that whether two selected services can be integrated together during composition in a way such that one can provide the necessary input required by the other to finally produce the desired output. If two services are functionally compatible then they are said to be *functionally dependent* on each other in a temporal order. If two services are causally dependent (the execution of one service requires input from the other service) on each other but are not compatible then the selection is said to be incompatible. This sort of incompatibility blocks any composition possibility after the services have been selected.

In the case of specification based modeling [81 - 106] the assumption that the requestor needs to know a particular specification language is not applicable for a lay user. Moreover, a standardized specification language for representing all service queries and all service descriptions has not been realized yet. Also specification matching is computationally expensive and can become NP-Hard in the worst case. Specifications, in

general, are thus used to study the behavioral validity and inconsistency of a service or a query.

3.10 Desire-based Query Modeling (DQM)

We now propose the *DQM* that contains all the essential information a user can easily provide and that can be satisfied in polynomial time in the worst case. Any kind of query has two essential parts that users can provide comfortably: (i) the *desire* (or required set of service output), and (ii) the (given) *input* information. There are several advantages of the proposed query model over task-based model and specification-based model. We describe each of them as below:

- ***Ease of Query Formulation***: The major difference between a *desire* and a task template or a specification is that a desire lacks the temporal and the functional dependency order between the desired services. Moreover, a *desire* is not a partially ordered set of service descriptions but rather a set of final states that the user needs from the system. For an example, in the car rental case study, the desire is the set {*car profile, car_rent confirmation*}. Suppose that the best car rental service can provide a confirmation but needs a third-party service that can generate a detailed profile of the car that has been rented. In this case it is not possible for the requestor to know that such a situation may arise and a third-party service has to be invoked. Hence, stating the query as a desire it becomes easy for a requestor to articulate his/her query.

- ***Improve Discovery Recall***: The given *input* of the query is very loosely coupled with the *desire*. This is so because the given input by the user does not necessarily map with the desired output states directly. For an example, a car rental service may require *age confirmation* as an input before it can produce the final desired output while the user may have given the input as {*name*, *source location, destination location*}. In this case we see that although the input is not directly mapable to the desired output as produced by the car rental service still there is a possibility that a third-party *ID verification* service may accept the given input and produce *age confirmation* as output that can then serve as the input to the car rental service. However, if the input and the desired output are strongly coupled as a task-template or a specification (i.e. only one kind of service is expected to take the given input and produce the desired output) then there can be cases of false negatives during discovery that can adversely affect the overall recall of the discovery process.

To explain the model in details let us take a case study as an example that we will adhere to for the rest of this dissertation.

Car Rental Scenario

Chris works for a consultant company in Kansas City. He desires to travel to Chicago from Kansas City on business within the next 14 days. However, he is busy on Tuesdays and Thursdays of every week and cannot travel. Here it is worth mentioning that Chris occasionally travels to neighboring city Manhattan to visit his mom. He wishes for a car rental service such that his desire is satisfied with minimum rental cost and maximum

comfort (that includes best weather conditions and nearest pickup location) without hampering his busy schedule. His personal preference is always weekends. Chris decides to lookup a car rental service that can best satisfy his needs. Chris is supposed to join office two days after his arrival to Chicago. Hence, he decides to fly back to Kansas City. He expects his chosen service discovery engine to take up his query and give back a confirmed car rental booking that contains the pickup location and the pickup date as the final output along with a confirmed flight ticket back to Kansas City as wished.

In this use case we can identify that the consumer (Chris) requires a rental car from his current place of location to Chicago. Therefore he needs a *car rental* service for this purpose. There can be several competing car rental services in the system. For an example, one service might have the description: {*O = (sedan confirmation, sedan information)*, *I = (customer age, customer ID)*}. Another may have the description: {*O = (SUV confirmation)*, *I = (customer social security, credit card information)*} while a third one may be having the description: {*O = (car confirmation, car information)*, *I = (customer name, customer age)*}.

Another requirement of Chris is that he needs to come back to this working place (which is Kansas City) after two days of his arrival to Chicago. Therefore, he needs a *flight booking* service as well. As with the car rental service there can be several available flight booking services as well. A service may have the specification {*O = (flight confirmation, itinerary information)*, *I = (customer name, source airport, destination airport)*}. Another service may have the description: {*O = (flight confirmation)*, *I = (customer credit card information, source city, destination city)*}. An interesting thing to note in this seemingly simple use case is that the current place of location of Chris is a variable (since it can be

both Manhattan and Kansas City). Hence, the car rental service should be aware of his current location so as to maximize his comfort by providing the nearest pick-up location. Another very interesting constraint is that the returning date to Kansas City solely depends upon the output of *car rental* service as it can be any of the next 14 days. Therefore until the car rental service is evoked the *flight booking* service cannot be executed by the engine. Hence, there exists a dependency between the two services that requires a dynamic composition. Other constraints are also added such as Chris's preference to drive in the weekends, his busy schedule on Tuesdays and Thursdays, and weather conditions. We now formalize the proposed query model as follows:

Definition 3.4: A *desire* is a set of 2-tuples $\{\langle O, R\rangle | \ O \neq null\}$ where O is the desired output state (set of output parameters) and R is the optional desired output effect on the system state set. ■

The *desire* component of a query is also called *Type-1 query* (or *Q-T1*)

Definition 3.5: An *input* is a set of 2-tuples $\{\langle I, R\rangle | \ I \neq null\}$ where I is the given input state (set of input parameters) and P is the optional desired pre-condition for the input to be valid. ■

The *input* component of a query is also called *Type-2 query* (or *Q-T2*).

Definition 3.6: A *query* Q is a 2-tuple $\langle desire, input\rangle \ni desire \not\equiv input$. ■

It is mentionable over here that the parameters that constitute both the set O and the set I are semantically defined within a collection of domain ontologies. Also a query can be initiated both by the user as well as intermediary services. We now propose three different

types of query as per the *DQM*: (i) *simple query*, (ii) *complex query*, and (iii) *compound query*.

Definition 3.7: A *simple query* Q^{SI} is a 2-tuple $\langle desire, input\rangle \ni \forall\langle O_i, R_i\rangle \in desire; |O_i| = 1$ ■

We can modify the use case query in order to form a simple query $Q^{SI} = D, I$ where:

$D = \{\langle\{confirmation\}, locatedIn\ (pickup\ location, Kansas\ City) \wedge destination\ (Chicago)\rangle\}$ and

$I = \{\langle name, source\ location, destination\ location\}, null\rangle\}$

Definition 3.8: A *complex query* Q^{CO} is a 2-tuple $\langle desire, input\rangle \ni \forall\langle O_i, R_i\rangle \in desire; |O_i| > 1$ ■

An example of a complex query is the query that is given by the requestor Chris in our use case. The query is structured as $Q^{CO} = D, I$ where:

$D = \{\langle\{confirmation, car\ info\}, locatedIn\ (pickup\ location, Kansas\ City) \wedge destination\ (Chicago)\rangle\}$ and $I = \{\langle name, source\ location, destination\ location\}, null\rangle\}$

The implication of the output parameters in a complex query is that all the desired output parameters (or states) are in conjunction. In the above query the desire is to have both the *confirmation* as well as the *car profile* together as a single desire. Absence of one of the parameters from the generated output of a matching service will disqualify the service from being a solution.

Definition 3.9: A *compound query* Q^{CP} is a 2-tuple $\langle desire, input\rangle \ni \exists\langle O_i, R_i\rangle, \langle O_j, R_j\rangle \in desire\ ; |O_i| = 1 \wedge |O_j| > 1$ ■

According to the above definition we can perceive a compound query as mixed set of simple and complex queries. For brevity we only consider queries that are complex while

proposing the service discovery process and the service composition process without any loss of generalization. We can do so because any compound query is essentially simple and complex queries that are OR-ed up during discovery and composition since each member in the query desire is independent from the requestor's standpoint.

3.11 DL-Encoding of DQM Query

Similar to service encoding we *DL-encode* queries into string of bits (1s and 0s) so that the semantic properties of the query parameters (both *input* and *desire*) are preserved. Queries that contain multiple desires are split into sub queries such that each sub query contains a single desire. Since the query model implies that multiple desires are in disjunction with each other (i.e. mutually independent) hence, each sub query can be treated and processed as a completely separate query. However, as mentioned earlier, at times there can be some sort of hidden dependency (like temporal) that is not discernible from the point of view of the requestor. For an example, suppose a query has one desire of renting a car from Kansas City to Chicago within next 14 days. Another desire within the same query is to return to Kansas City from Chicago after spending a week in Chicago. The Q-T1 of this query is

$\{\{\{car\ profile, car\ confirmation\}, \{locatedIn(pickup, Kansas\ City) \wedge destination(Chicago) \wedge differenceLessThan(trave\ date, current\ date, 15)\}\rangle \ldots \langle\{flight\ confirmation\}, \{destination(Kansas\ City) \wedge differentEqualTo(travel\ date, current\ date, 21)\}\rangle\}$

In this example we see that the hidden dependency between the two desires is that if the first desire is not satisfied then the second desire even if satisfiable becomes unnecessary. Hence, even if the two desires can be treated as two sub Q-T1s still there lies a hidden problem. We leave the treatment of these cases as a future work. For now, we assume that multiple desires do not have any hidden dependency and are purely disjunctive in nature. Based on this assumption, each of the sub queries are then encoded separately as follows:

- ***Simple Query Q-T1 Encoding***: Encoding simple query is pretty straightforward. Since a simple query contains only one desire parameter the Q-T1 gets the bit code of the single concept parameter that it carries. For an example, if the desire is {*car*} then according to the ontology encoding shown earlier in this chapter the Q-T1 is encoded as *DL-code*(*car*) = 0^*10011.
- ***Complex Query Q-T1 Encoding***: Encoding a complex query is done by OR-ing up the codes of each of the individual desire parameters within the Q-T1. Hence, if the desire is {*car, confirmation*} then Q-T1 is encoded as: [*DL-code*(*car*) ∨ *DL-code*(*confirmation*)].
- ***Compound Query Q*-T1 *Encoding***: Since a compound query is a disjunctive collection of simple and complex queries hence, we can split it up into simple and complex sub-queries. After that we can encode each of the sub queries as stated in the earlier two cases.
- ***Q-T2 Encoding***: Irrespective of the type of query the Q-T2 component is encoded by OR-ing up the individual codes of input concept parameters. Thus, for an example, if the

input is {*name, source city*} then the Q-T2 is encoded as: [*DL-code*(*name*) ∨ *DL-code*(*source city*)].

3.12 g-subsumption based Query Matching

Semantic query matching for service discovery [145 - 154] is most often based on the web service match model proposed by Paolucci et al [132] and Sycara [133]. According to this model there are three types of query matches: (i) exact, (ii) plugin, and (iii) subsume. *Exact matching* is the case when all of the desired input/output set are either same or direct subclasses of one or more generated output of a matching service. For an example, the desire of a *car* can be considered as an exact match of a service providing *vehicle* according to the vehicle ontology introduced earlier in this chapter. *Plugin matching* is the case when all of the desired output set are indirect subclasses of one or more generated output. For an example, the desire of a *sedan* can be considered as a plugin match with a service providing *vehicle*. *Subsume matching* is the case when all of the desired output set are super classes of one or more generated output. For example, the desire of a *car* can be considered as a subsume match with a service providing *sedan*. The relative strength of these three types of matches is given as: *subsumes* < *plugin* < *exact*.

Although the above query matching scheme is widely followed in research proposals on semantic service discovery we contest this scheme to be seriously flawed in several aspects. In this section we first identify and discuss these flaws. Then we propose a derived matching scheme that eliminates the identified flaws. The flaws are identified as follows:

A. ***Order is incorrect***: For Q-T1 query (containing *desire*) the match is computed over the service output. In this case any service providing an output state that is either exact or a hyponym of the desired output state should be selected as one of the candidate end services. For an example if Q-T1 desires the *car* then a rental service producing *sedan* is a match. This is because under all situations the service promises to provide *sedan* which is a *car*. Hence, the probability that the Q-T1 is going to be satisfied is 1 (assuming that the pre-condition of the service is satisfied by Q-T1). On the other hand, suppose the service has output *vehicle* then the match cannot be considered strong (although it can still be considered a weak match). This is because the output of the service may take on several possible discrete states one of which is the state *car* (other states can be *bus, boat, aeroplane, etc*). Thus, the probability of that the Q-T1 query can be satisfied is dependent on the number of discrete output states that the service can produce. Hence, for Q-T1 query we argue that the order of match strength (*query to service*) should be: *exact* > *subsumes* > *plugin*.

B. ***Order is not preserved*:** For Q-T2 query (containing *input*) the match is computed over the service input. In this case any service that requires an input state that is either exact or a hypernym of the given input state should be considered as a Q-T2 match. For an example, suppose a service requires *car* (type) as input and produces *manufacturer* as output. If the given input of a Q-T2 query is *sedan* then the service should be considered as a match. However, if the given input is *vehicle* then the service may be a match with a match probability dependent on the number of discrete given input states possible. In this case the given input can sometimes be *car,* while at some other time be *bus*. Hence,

for Q-T2 we argue that the order of match strength (*query to service*) should be: *exact* > *plugin* > *subsume*. Therefore, we conclude that the match order is not preserved and differs according to the query type.

C. ***Accuracy can be improved:*** The matchmaking algorithms that are designed according to the Paolucci match order suffer from serious accuracy issues. This is because according to these algorithms a match is valid if all the desired output parameters have match with one or more service output. To illustrate this further let us consider a complex Q-T1 query having a desire {*car profile, confirmation*} and a service having output {*confirmation*}. Now, as the desire is not completely satisfied hence, in conventional models the service will not be chosen as a candidate. However, in reality it may be possible that service may call up another third-party service that can generate the output {*car profile*} when given the input {*car*} by the service. Thus, considering only complete match may adversely affect the recall. In the same way if the Q-T1 query is simple having multiple desires then also the recall will be negatively affected. Also since the order is incorrect for both Q-T1 and Q-T2 queries hence, the algorithms will adversely affect the precision by falsely including services as strong matches (while in reality they may be weak).

In our proposed matchmaking scheme we do not apply the same match strength order for Q-T1 and Q-T2 queries. There are 3 basic types of matches: (i) *strong*, (ii) *weak*, and (iii) *sibling*. A strong match is either an exact match or a plugin/subsume (depending on query type) match (*car vs. sedan*). A weak match is a subsume/plugin (depending on query type) match (*sedan vs. car*). Lastly, a sibling match is neither a strong match nor a weak

match but a match where there is at least one most specific common parent (*bus* vs. *car* where both has most specific parent *vehicle*). Based on these 3 basic matches we consider 9 different match possibilities between a query and a service. They are given as per strength as follows:

1. ***Strong-Strong Match (SS)***: This is the case when the *desire* part of the query (Q-T1) has a strong match (*exact* or *subsume*) with service output while the *input* part of the query (Q-T2) has a strong match (in this case *exact* or *plugin*) with the required input of the service.
2. ***Strong-Weak Match (SW)***: This is the case when the *desire* part of the query (Q-T1) has a strong match (*exact* or *subsume*) with service output while the *input* part of the query (Q-T2) has a weak match (in this case *subsume*) with the required input of the service.
3. ***Strong-Sibling Match (SSi)***: This is the case when the *desire* part of the query (Q-T1) has a strong match (*exact* or *subsume*) with service output while the *input* part of the query (Q-T2) has a sibling match (in this case *sibling*) with the required input of the service.
4. ***Weak-Strong Match (WS)***: This is the case when the *desire* part of the query (Q-T1) has a weak match (*plugin*) with service output while the *input* part of the query (Q-T2) has a strong match (in this case *exact* or *plugin*) with the required input of the service.
5. ***Weak-Weak Match (WW)***: This is the case when the *desire* part of the query (Q-T1) has a weak match (*plugin*) with service output while the *input* part of the query (Q-T2) has a weak match (in this case *subsume*) with the required input of the service.

6. ***Weak-Sibling Match (WSi)***: This is the case when the *desire* part of the query (Q-T1) has a weak match (*plugin*) with service output while the *input* part of the query (Q-T2) has a sibling match (in this case *sibling*) with the required input of the service.
7. ***Sibling-Strong Match (SiS)***: This is the case when the *desire* part of the query (Q-T1) has a sibling match (*sibling*) with service output while the *input* part of the query (Q-T2) has a strong match (in this case *exact* or *plugin*) with the required input of the service.
8. ***Sibling-Weak Match (SiW)***: This is the case when the *desire* part of the query (Q-T1) has a sibling match (*sibling*) with service output while the *input* part of the query (Q-T2) has a weak match (in this case *subsume*) with the required input of the service.
9. ***Sibling-Sibling Match (SiSi)***: This is the case when the *desire* part of the query (Q-T1) has a sibling match (*sibling*) with service output while the *input* part of the query (Q-T2) has a sibling match (in this case *sibling*) with the required input of the service.

Each of the above 9 match cases may be either partial (where not all the desire set is satisfied by a single match) or complete (where all of the desire set is satisfied). It may happen that in some cases there is a mixed match. For an example, suppose a complex query has desire {*car profile, email confirmation*} while input {*name, source location, destination location*}. A service has output {*sedan profile, sms confirmation*} while input {*name, ID, destination location*}. In this case, the Q-T1 has a mixed match with the service output where the first desire state has a strong match while the second desired state has a sibling match. In such a situation the weaker match dominates and the desire is said to have a complete sibling match. Similarly, for the Q-T2 the required input is said to have a partial strong match. Note that Q-T2 is always matched with respect to the service while Q-T1 is

always matched with respect to the query. We now present the matchmaking algorithm that is applied for pair-wise query-service matching.

3.13 2-Phase Service Discovery Algorithm: Outline

Service queries are shot into an SOA based system in two phases: (1) *desire matching phase* and (ii) *input matching phase*. The proposed discovery & composition system (both broker based as well as agent-based SMARTSPACE) splits the given query into the *desire* and the *input* components as a pre-searching process. Each component is treated as a separate query: *type-1 query* for the desire matching phase (Q-T1) and *type-2 query* for the input matching phase (Q-T2).

- **Desire Matching Phase**: The first phase called *desire matching*. In this phase the system works for the requestor and looks for all services that can provide either partially or completely his/her desire. At the end of this phase a set of matching candidate services (called *end services*) are retrieved that are capable of satisfying the query desire.
- **Input Matching Phase**: If the retrieved candidate service set is empty then the second phase is aborted. If not then the second phase, called *input matching*, starts off. In this phase the system shifts mode and starts working for the candidate service set by trying to serve them with their required input. It is at this phase that the algorithm differs from the proposed centralized broker based model to the proposed distributed multi-agent based model (i.e. SMARTSPACE). More details will be given in subsequent chapters.

3.14 Discussion

The advantages of query modeling have already been discussed in the previous section. In this section we conclude the chapter with a discussion on the effect of 2-phase service searching and query type classification into simple, complex and compound query on service discovery and composition.

A. **Effect on Service Discovery**:

- ***Improves Matching Computation***: The advantage of 2-phase search on service discovery can be significant. One of the properties of the proposed query model states that if Q-T1 query during first phase of service searching (*desire matching phase*) fails (i.e. candidate set is empty) then there is no need of conducting the second phase (*input matching phase*). This is because failure of type 1 query suggests that there is no *end service* that can produce the desired final state. In such a situation the second phase is unnecessary. This significantly reduces unnecessary computational overhead on the system for a lot of query types that are not serviceable. When compared to prevalent single phase service discovery based on task-based or specification-based queries we can observe that in those techniques a lot of unnecessary matchmaking computation is done by considering all the description parameters (service name, input/output/pre-condition/result parameters, service category, QoS parameters, etc) in cases where the many of the services cannot output the desire. In other words, in our model the failure cases are detected much faster.

- ***Improves Search Precision***: Since the query is classified formally into simple, complex, and compound queries based on the nature of the *desire* component the service discovery can be done with higher precision as compared to task-based and specification-based queries. This can be illustrated using the use case scenario. Suppose that the car profile service can only provide the *car profile* but cannot provide the *confirmation* that was desired as well by the requestor. Also let us suppose that the "car information service" is provided with all the input that it requires (viz. *name*). In a task-based model, since the query matching is integrated hence, the desired abstract service will have a high match with the car rental service although an important desire (i.e. *confirmation*) is missing. This results in false positive and affects the search precision. However, in our proposed query model as the searching is done in two-phases, hence the system reports a partial match with the car information service. As the query is a complex query hence, the system rejects such a partial solution since the *car profile* output is in conjunction with the desired *confirmation* output. However, had the query been a simple query consisting of two independent desires (*car profile* and *confirmation*) then the partial solution would have been accepted and suggested by the system. Thus, we see that query classification helps in increasing the precision in our proposed model. More details of the discovery process (both broker based as well as SMARTSPACE based) will be given in later chapters.

B. **Effect on Service Composition**:

- ***Eliminates Issue of Selection Incompatibility***: 2-phase service searching allows seamless integration of service composition with service discovery and selection. Treating the composition problem separate from the discovery and selection problem has several disadvantages. One of the major problems is the *issue of selection incompatibility* if the selection process is done before composition. In our proposed model, since the first-phase involves only discovery while the second phase involves the selection process integrated into the composition process hence, the 2-phase service search helps in avoiding the aforesaid problem.
- ***Improves Composition Computational Complexity***: The other problem that is introduced as a result of this approach is that because of the possibility of a large number of functionally incompatible selected services (especially in the case when the composition involves the integration of lot of services) hence, in the worst case the problem becomes NP-Hard. In our proposed solution the explosion of selection incompatibilities is avoided using two different techniques applied specifically for the centralized and the distributed agent-based systems. For the centralized model a notion of *reachability* is applied using which a set of candidate *source services* are retrieved in the second search phase that take in the given input and are composable (i.e. reachable from) directly or indirectly (using *intermediary services*) with the set of already retrieved candidate *end services* in the first phase of the search. For the distributed model since there is no centralized global knowledge of the underlying service dependency network hence, the notion of reachability may become quite

complicated and expensive to be implemented. Instead, candidate services (starting from *end services*) shoot Q-T1 queries where desire becomes the required input that they could not find or sometimes in the cases of partial satisfaction of initial desire in complex queries the Q-T1 carry desires that are still to be satisfied. In both the proposed centralized and distributed systems, since the selection process is deferred till the very moment of composition hence, the issue of functional incompatibility does not arise. A detailed discussion on service composition is given in later chapters on service composition.

3.15 Results

To get an insight over the computational validity of the proposed *DL-Encoding* technique and its effect over the *g-subsumption* algorithm we have conducted certain experiments. We conducted our experiments on a system having a CPU cycle of 1.4 GHz and a RAM of 2 GB. The experiments had two objectives as follows:

- To understand the computational efficiency of pre-processing g-Arrays (i.e. *DLEncoding*) of semantic web services (collected from standard OWLS-TC v2 dataset) using *DL-Encoding*.
- To compare the computational efficiency of *g-subsumption* with other DL-based reasoners: FACT++, Pellet, and Racer.

For the first experiment we first folded the ontologies covered in OWLS-TC v2 into several sub-ontologies – each with different concept sizes starting from a concept size of 100 to a concept size of 2900 (figure 14). We found that there was an approximately

linear growth in the computational cost of *DLEncoding*. We conducted the same experiment with a randomly generated set of ontologies and found a quadratic growth rate instead (figure 15). Since the probability of inclusion of complex concept definitions increases with the ontology size in the randomized setup hence, the computational growth rate is more in this case as compared to OWLS-TC dataset.

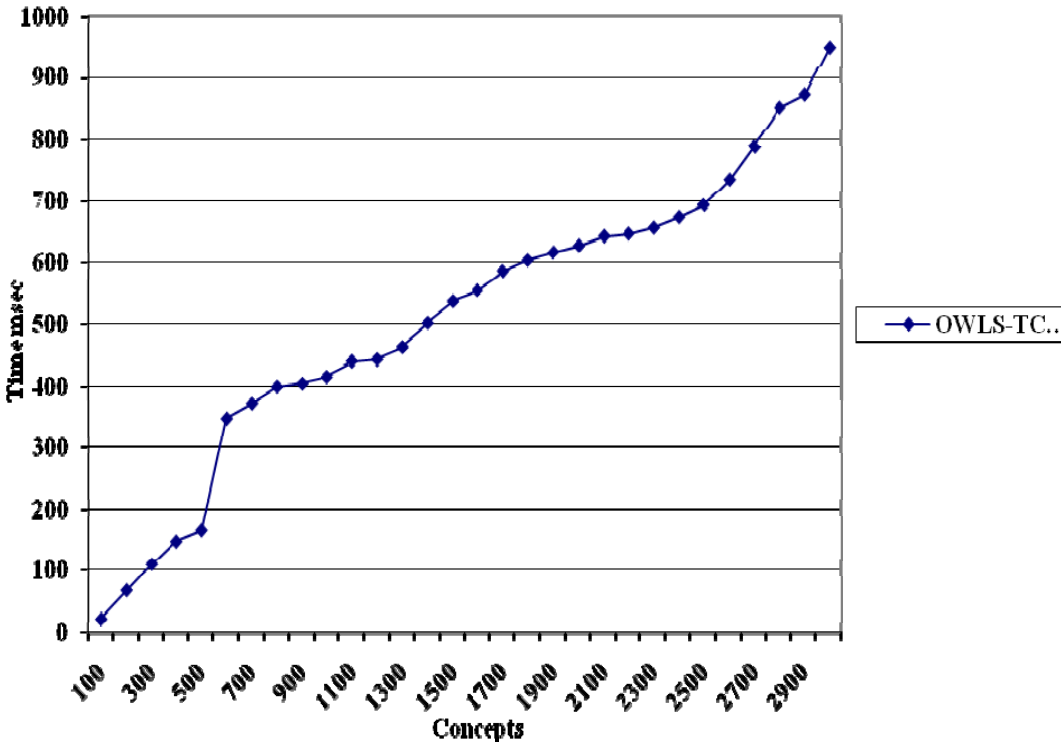


Figure 14: DL-Encoding Runtime over OWLS-TC v2 dataset

For the second experiment we again took the standard OWLS-TC v2 dataset. The concepts within the ontologies are pre-processed into their corresponding *O-arrays*. The *O-array* represents the most significant service feature - *Output*. We then conducted a pair-

wise subsumption testing between all distinct *O-arrays* within the ontologies for each of the subsumption reasoners (vi. Pellet, FACT ++, Racer, and *g-subsumption*). We averaged the computational time for subsumption as shown in figure 16. We observed a major improvement in the subsumption test runtime for *g-relation*. The drop in the runtime is because of two reasons: (i) the bit based *OR* operation, and (ii) the stratified and compact

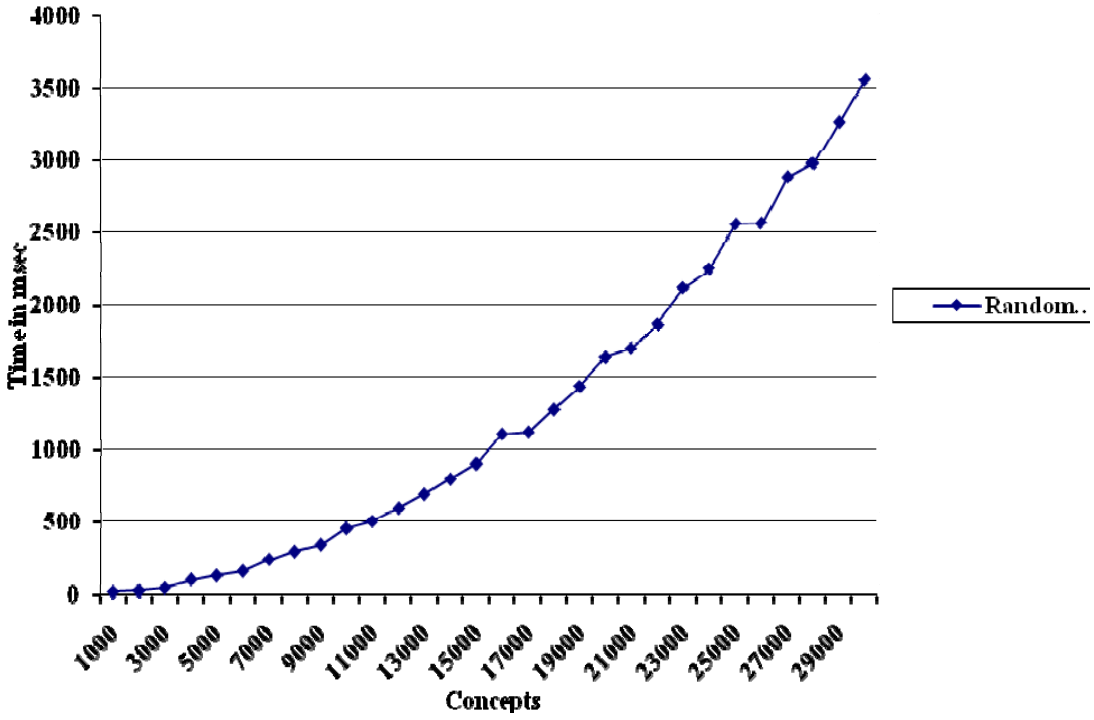


Figure 15: DL-Encoding Runtime over Random Dataset

form of the *g-array* data structure as compared to a full DL definition taken as input by the DL-based reasoners. We also observe that the dynamic *g-relation* performance is not as good as the static *g-relation*. This is because the static subsumption is basically based on the *b-code* that uses the normal logical OR operator while the dynamic subsumption is based

on the *DLcode* that uses the *DL-OR* operator which has a significant constant factor in its worst case time complexity.

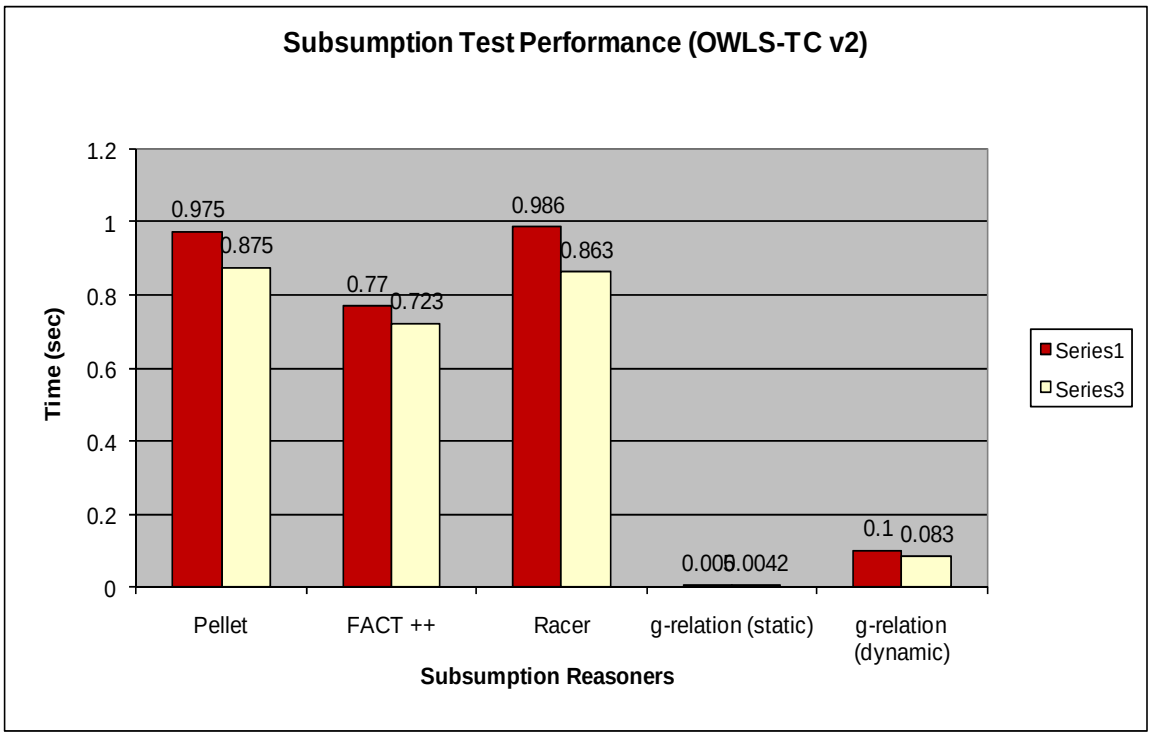


Figure 16: Comparative Analysis of DL Subsumption Test Runtime

3.16 Conclusion

The chapter has introduced the *g-subsumption* service matchmaking algorithm. The proposed algorithm is a linear time matchmaker that does not involve DL-reasoner based subsumption while checking subsumption type for service descriptions that are dynamically defined by service providers. For this purpose it utilizes a novel bit-based encoding technique called *DL-Encoding*. Service matchmaking is going to be one of the foundational operations for efficient discovery of services in two ways:

- Organizing services into their corresponding (functional) categories for pruning the search space
- Mapping consumer queries into the organized service space to discover the best matching services.

While the former will be discussed in the next chapter the latter is going to be treated in chapter 5 and 6.

CHAPTER 4

SERVICE ORGANIZATION BY LEARNING SERVICE CATEGORY

4.1 Introduction

As mentioned in the previous chapter one of the major operations in Service Oriented Architecture (SOA) based systems is service discovery. In order to facilitate dynamic on-demand access to services we need an efficient way of discovering the required services out of a large pool of functionally different services. Service discovery process can become very efficient when service advertisements are categorically organized into well-defined access structures such as Universal Description Discovery & Integration (UDDI [8]) and Distributed Hash Tables (DHT) based registries such as CHORD [155]. A conventional way of grouping services into categories is to apply machine learning techniques for learning service categories. The process is called *service category learning*. The general problem statement is as follows:

Given a set of service descriptions (that contain functional and other QoS features) we need to model a learner that can predict the labeled or unlabeled category (mostly functional) of the services by observing their corresponding descriptions with minimum prediction error.

Functionally similar category classes/clusters can then be indexed into centralized registries or into DHTs depending upon the application, domain, and underlying network overlay issues. A consumer query is then mapped over the cluster space and the service having the best match is selected. The evaluation of a best match is usually based upon a

pre-defined ranking function and the process is called *Service Selection*. Some researchers have used supervised learning algorithms such as SVM based classifiers [165 - 166], LSI (Latent Semantic Index) based classifiers [164], Probabilistic LSI based classifiers [172 - 173], and naïve Bayesian classifiers [157, 163] to classify service advertisements (formally represented by a description language such as WSDL or OWL-S) into domain categories according to the service functionality. Ensemble learning has also been proposed by some work [161, 163]. Some other works have applied partitional unsupervised learning techniques such as KNN (k-nearest neighbor) [163], k-means (and derivatives) [168, 176], star clustering [169] etc. There are also research works that have applied hierarchical unsupervised learning algorithms such as Agglomerative (and derivatives) [163, 171, 181], Word-IC algorithm [177], etc. In most of these works a service description is formalized as a feature vector that constitutes the functional features (i.e. *Input*, *Output*, *Pre-condition*, and *Result*) and descriptive features (usually key terms having high TF-IDF).

In this chapter we propose a novel service category learning algorithm, called *Semantic Taxonomical Clustering* (STC), which utilizes semantic descriptions of services. However, the approach does not apply statistical learning techniques (as is the case in most service category learning algorithms). The assumption behind this approach is that service descriptions have to be semantically defined using a set of domain ontologies (see chapter 3). The chapter starts with a detailed discussion on the limitations of some significant statistical service category learning approaches. Since all these approaches are based on the notion of a distance measure that models the pair-wise similarity between two services the chapter then shows some of the limitations of distance measure based approaches in general.

After that the STC algorithm together with its conceptual foundations and properties is proposed and analyzed. The chapter concludes with a comparative empirical result that shows: (a) the runtime efficiency of the proposed algorithm as compared to a nearest neighbor based clustering algorithm (designed by us) over a set of randomly generated dataset and (b) the domain-based accuracy of STC when compared against an expert evaluated categorization of the standard OWLS-TC v2 web service dataset.

4.2 Related Work

There have been several research works on service category learning so far [156 - 177]. Service categorization is usually motivated by the thematic properties that have been proposed in standards such as UNSPCS (United Nations Standard Products and Service Codes) [178], NAICS (North American Industry Classification System) [179], etc. Thematic properties may include the service functional properties (*input* types, *output* types, *pre-condition*, and *result*), the QoS properties (availability, reliability, etc), domain information (i.e. area of application) that can be extracted out of service descriptions. A distance measure is then modeled that is used to compute the pair-wise similarity between two service descriptions. In general the distance measure can be of two types: (i) keyword based and (ii) ontology based. In key word based distance measures the similarity of two services is computed based on TF/IDF technique derived from IR research [147 - 148, 158, 170 - 171]. TF/IDF is done to ascribe weight to the documents (service descriptions) with respect to a particular term (attribute keyword). The weighted attributes (functional attributes are input, output, pre-condition, post-condition) of the services are represented as a feature vector and

then the similarity between the attributes are computed based on conventional vector cosine similarity measure. IR based similarity computation can be very useful where we do not have formalized semantics for the service descriptions. As an alternative approach, there has been significant research on ontology based semantic distance measure [132, 153 - 154, 180, 182]. For any service similarity model we need to define a measure. Semantic distance measure can be classified into three categories: (i) taxonomic distance based [183 - 184], (ii) information content (IC) based [185 - 186], and (iii) concept property based [182, 187 - 188]. There also have been considerable researches on hybrid approaches incorporating features from one or more measures.

In most research works relating to service categorization the learning problem is limited to functional properties only. The motivation for this approach is that service category learning is primarily done for service discovery and discovery is essentially functional matching consumer queries with services. In general we can classify all such learning techniques into two basic learning frameworks: (i) supervised learning and (ii) unsupervised learning. In supervised learning mode it is essential to have a sufficiently "sound" training data of service descriptions that guarantees minimum over-fitting and under-fitting. Also, we need to have a clear understanding of the categories into which new service descriptions can be fitted into. Research works have involved classical Machine Learning techniques such as SVMs (Support Vector Machine) [165 - 166] and NBC (Naïve Bayesian Classifier) [157, 163]. Some works have also used more recent Information Retrieval models such as LSI (Latent Semantic Index) [164] and PLSI (Probabilistic LSI)

[172 - 173]. However, in general, there are some major drawbacks in supervised learning of service categories:

- ***Curse of Service Category***: In an open SOA based system the number of classes that needs to be predefined is practically impossible to estimate. This is because old categories (i.e. classes) get deleted non-deterministically and new categories get added non-deterministically. Hence, a suitable training set is very difficult to create. The problems of over-fitting and under-fitting are always innate for service categorization. Hence, service category algorithms such as Naïve Bayes and SVM may not be good choices (see experimental results section in this chapter for supportive evidence).
- ***Curse of Dynamic Service Set*:** Since the set of services is dynamic hence, we cannot use learning algorithms that are not online. Online statistical learning is computationally expensive in terms of cluster space convergence as the training period can be very long. Also, it is very difficult to guarantee a convergence for a particular algorithm.
- ***Problem with LSI*:** LSI (Latent Semantic Indexing) is a technique that helps to reduce the dimensionality of service-term matrix into *k-topics* such that services that do not have common terms can also be brought together into same group if they have the same topic. In this context a *topic* can be seen as a thematic class of a service while terms are the parameters and descriptive terms. The intuition is that if the matrix can be decomposed into an SVD (Singular Vector Decomposition) then the original sparse matrix can be reduced into a lower *k*-dimension matrix using *k* ranks. However, the selection of the optimal k value is not easy. Also, the approach is intuitive and is not theoretically sound.

- ***Problem with PLSI*:** An advanced optimization over the LSI approach is PLSI (Probabilistic LSI). In this approach the document, topic, and term are structured as a Bayesian Network instead of a matrix. However, one of the big disadvantages of this approach is that it is not easy to estimate the optimal number of latent variables (i.e. topics) required to model the best classifier. Another major drawback is that topics are assumed to be conditionally independent with respect to service descriptions within the Bayesian network structure. However, this assumption may not be valid since the existence of one topic may influence the occurrence of another topic in cases where services may belong to multiple topics (e.g., a car rental service can act as both a rental service as well as a car lookup service).
- ***Problem with Naïve Bayesian Classifier***: In category learning using Naïve Bayesian classifier the disadvantage is the lack of proper statistical data for dynamic service set. Since in order to calculate the posterior probability for a service to belong to a particular category class we need to estimate the prior probability of the service and the likelihood that the category class accurately includes the service. To have accurate estimation (generally based on frequency of occurrence) we need a strong representative statistical data set which is difficult to obtain. Moreover, the algorithm is not suitable for online learning.

Unsupervised learning mode is the other alternative method of service category learning. In this mode we do not need to have pre-understanding of the service categories into which services have to be fitted with. The training dataset is used to generate a learning model that is basically a set of clusters of service descriptions such that it maximizes the

global inter-cluster distance (i.e. service functional dissimilarity) while minimizes the global intra-cluster distance. In other words, services are grouped into functionally similar clusters in a way such that new observations do not disturb the cluster space topology by re-modeling the learner (creating new clusters or modifying old clusters). One of the commonest techniques in this direction of service category learning is using k-means based algorithms [168, 176]. These algorithms are partitional in nature in the sense that it partitions the training set into disjoint partitions (i.e. clusters) where the number of partitions is pre-estimated. Subsequent new service descriptions are then fitted into these partitions with a minimum error. Another technique of unsupervised learning is to use hierarchical clustering algorithms such as agglomerative based clustering [163, 171, 181]. In these approaches service descriptions are pair-wise compared to form of a kind of minimum spanning tree over the cluster space (instead of disjoint partitions). The tree structure enforces a partial ordering over the cluster space by representing nodes at lower depths to be a more generic set of similar services while nodes at higher depths are more specific sets of similar services. The partial order is essentially the intra-set distance that is lower in bottom level nodes while higher in top level nodes. In this approach there is no requirement to pre-estimate the total number of nodes (i.e. sets/clusters) as they are self-induced by the algorithms. As with supervised learning mode unsupervised service category learning also comes with several drawbacks as follows:

- ***Problems of Partitional Clustering***: Partitional clustering algorithms such as k-means and derivatives [168, 176] have some open problems that still demand efficient solution. One such problem is the estimation of the number of centroids (i.e. clusters) that can

converge to a optimal cluster space. The other problem is the optimal selection of centroids as different selections lead to different qualities of final cluster space (in terms of intra-cluster similarity and inter-cluster dissimilarity). The third problem is that since such algorithms are ad-hoc it is not suitable for a services particularly which need dynamic online clustering

- ***Problems of Agglomerative Clustering*:** As agglomerative clustering of services [163, 181] is a bottom-up ad hoc approach hence, it is computationally expensive as it requires all pair similarity computation. Also there needs to be a halting condition for the merging which is difficult to design. The halting condition should be such that the best level of the hierarchy is achieved with maximum precision and recall. Also, since the approach is ad-hoc we cannot apply it in its entirety to dynamic service set clustering.
- ***Problems with Merge-Split Clustering*:** In [171] an agglomerative merge-split clustering (called Woogle) has been proposed for service discovery. The approach is slightly different from most approaches in that the clustering is not done over the service set but is done over the set of terms. A similarity metric is proposed based on term association. Two terms are considered to be similar if they have a sufficient *confidence-support* score. Each cluster finally contains a set of *kernel terms* that enforces strong intra-cluster similarity. A cluster is split so that each cluster can have all its member terms as kernels. Services are discovered based on Input-Output matching with a query where Input and Output is a bag of terms. If the required output terms can be clustered into the same clusters that represent the generated output term bag then there is a match. However, the major drawback of this approach is that the cluster space convergence is computationally

expensive. Also since it is agglomerative in nature hence, it is not suitable for dynamic service set clustering.

- ***Problems of Star Clustering*:** Categorizing services using star clustering has been introduced in [169]. However, the approach suffers from the problem of ad-hoc pair-wise similarity computation that is essentially expensive. Moreover, since it is ad-hoc hence, it is not suitable for dynamic service set clustering. Apart from that, identifying a star structure within the linked up similar services is also computationally expensive.
- ***Problems of KNN***: KNN (k-Nearest Neighbor) [163] can be a good choice for online clustering. However, it may suffer from under-fitting because of the unknown randomness within the dynamic service set. Also since it is an all inclusion approach with no revision of existing clusters for splitting consideration the precision can be adversely affected.

4.3 Shortcomings of Distance-based Learning

In general, most service category learning techniques (supervised and unsupervised) discussed so far are distance measure (i.e. similarity measure) based. In this section we discuss some of the major limitations of distance-based learning:

- ***Problem of Threshold Selection*:** Threshold selection is necessary for learning algorithms that require some sort of selection condition for two services to be considered similar. Barring a few techniques (such as k-means and k-nearest neighbor based algorithms) most service category learning algorithms require an optimal threshold selection. If the threshold is too tight then it might affect the recall while if the threshold

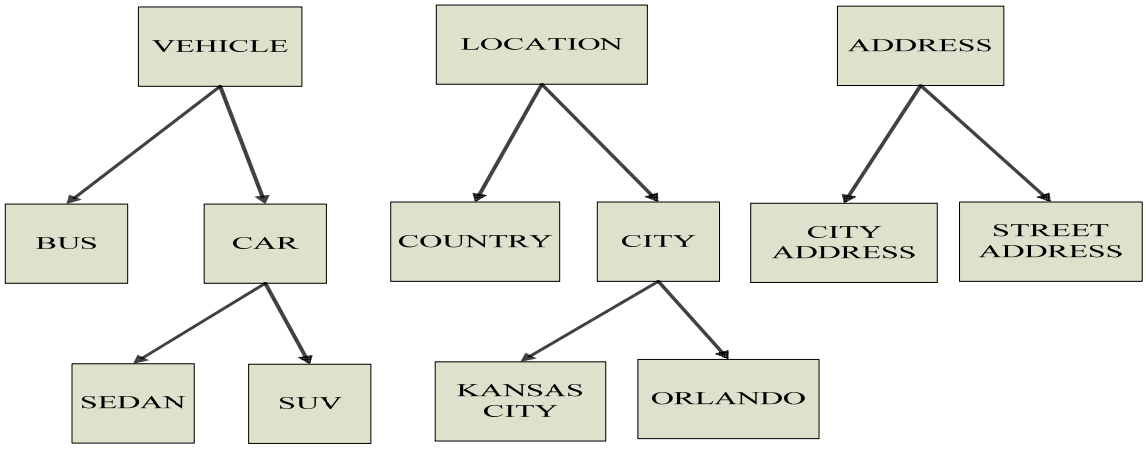


Figure 17: Ontology of 3 Taxonomies: *Vehicle, Location, & Address*

is too loose then it might affect the precision. In most cases the choice of threshold is empirically done. This consumes the overall learning time period and requires lot of manual intervention.

- ***Problem of Sample Selection Order for Online Learning*:** In online learning mode we do not have a fixed service set to begin with. In such a framework the order in which services are observed and categorized may have negative side-effect on the overall clustering performance. We call this problem of sample selection order. To explain this problem we take an example. Let us consider three services s_1, s_2, and s_3. Let these services need to be clustered according to their output feature O (stratified clustering). It is given that $s_1.o$ = {*car, location*}, $s_2.o$ = {*vehicle, city*, *address*}, $s_3.o$ = {*SUV, street_address*}. The domain set for this example is: *{vehicle, location, address}*. Semantically, s_1 and s_2 are siblings under a common abstraction {*vehicle, location*} while s_3 is sibling to this abstraction under a common abstraction {*vehicle*}.

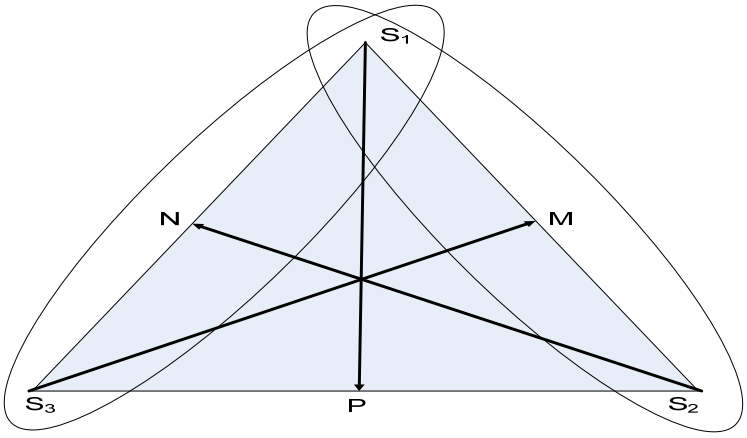


Figure 18: Effect of Sample Selection Order over Euclidean Space

In other words, all the three services should belong to one categorical cluster. Suppose that the temporal online order in which each of the three services are observed into the system (i.e. the timeline over which they are first published) is: $s_1 \rightarrow s_2 \rightarrow s_3$. We now prove that the converged cluster space can be dependent on the *sample selection order*. It is to be noted that *sample* means a service in this context.

Theorem 4.1: Given three sample points s_1, s_2, and s_3 in a sample space S (where S is a metric space) there exists some selection threshold d s.t. $CS^{123} \not\equiv CS^{132} \not\equiv CS^{321}$ where CS^{xyz} represent the cluster space formed as a result of sample selection order $x \rightarrow y \rightarrow z$.

Proof: We can draw a triangle $\Delta s_1 s_2 s_3$ on the metric space S (figure 18) representing the sample space. Now, considering order $s_1 \rightarrow s_2 \rightarrow s_3$ let us assume that $dist(s_1,s_2) < d$. We see that S_3 is included into the cluster $\{S_1, S_2\}$ if and only if $dist(s_3, M) < d$ *(where M is mid-point of* $s_1 s_2$*)*. Using Apollonius Theorem [190], $\overline{s_3 M}^2 = \frac{\overline{s_3 s_1}^2 + \overline{s_2 s_3}^2}{2} - \frac{\overline{s_2 s_1}^2}{4}$.

Similarly, for the sequence $s_1 \rightarrow s_3 \rightarrow s_2$, we get $\overline{s_2N}^2 = \frac{\overline{s_2s_1}^2 + \overline{s_2s_3}^2}{2} - \frac{\overline{s_1s_3}^2}{4}$ (*where N is the mid-point of* S_1S_3). Now $\overline{s_3M}^2 = \overline{s_2N}^2 iff \left[\frac{\overline{s_3s_1}^2 + \overline{s_2s_3}^2}{2} - \frac{\overline{s_2s_1}^2}{4} = \frac{\overline{s_2s_1}^2 + \overline{s_2s_3}^2}{2} - \frac{\overline{s_1s_3}^2}{4}\right]$. This means that $\overline{s_3M} = \overline{s_2N}\ iff\ s_3s_1 = s_3s_2$. Thus, unless the three sample points are topologically equidistant, the distance between the third selection and the first cluster will vary based upon the particular sequence chosen. Hence, for a threshold *d* it may happen that $\overline{s_3M} > d > \overline{s_2N}$ *or* $\overline{s_3M} < d < \overline{s_2N}$. From this result we can conclude that $CS^{123} = CS^{132} = CS^{321}\ iff \nexists d\ \ni\ \forall i,j,K; d \in \left[\overline{s_iK}, \overline{s_jK}\right]$ where $K = \{M, N, P\}$ is the mean sample point of first cluster. It is to be noted that the above theorem can be generalized for *n* sample points.

- ***Problem of Disjoint Category Assumption*:** In most distance measure based learning approaches the basic assumption underlying the learning problem is that clusters/classes are disjoint. In other words, if a particular service belongs to one cluster then it cannot belong to another cluster within the same cluster space. However, this assumption is not applicable for services since a service can actually be categorized in more than one way. For an example, a car rental service having *Output = {car_profile, confirmation}* can also be used as a car lookup service without the confirmation.
- ***Problems of Integrated Similarity Measure*:** Integrated Similarity measure based learning is a very popular approach where the measure is linear weighted combination (mostly a simplex) of all the functional service features (*Input*, *Output*, *Pre-condition*,

Result) so as to produce a single distance score. However, integrated measures suffers from some serious drawbacks that have been discussed below:

- ***Effect of I/P match over O/R match***: In integrated approach it may happen that two service vectors are exactly similar in terms of their *Input (I)* and *Pre-condition* (*P*) features while different in terms of their *Output (O)* and *Result (R)* features. A high match in *I* and *P* can shift the overall similarity score beyond the chosen threshold even if there is a low match in *O* and *R*. This effect can be reduced to some extent by carefully choosing weights for each of the features [182]. However, this method also does not guarantee the elimination of this problem for all cases. For an example, in a case where the number of *Input* parameters of the two compared services is significantly higher than the number of *Output* parameters and there is an exact match between their *Input* we may see that the two services are clustered together as similar. Thus, two services may be incorrectly clustered together even though their output and effect are different. However, service functionality is directly dependent on the *O feature* and its corresponding *R* feature. The *I* and *P* features indirectly determine the *O* and *R*. If any two services have the same output then irrespective of the type of input we can say, broadly, that they are functionally similar to some extent. In the given example s_1 and s_2 have output {*vehicle, location*}. Imagine that s_1 takes as input {*customer_name, credit_info*} and s_2 takes input {*customer_password*}. Thus, semantically the *I* feature in this case has no subsumptive match. However, s_1 and s_2 are functionally similar as they do the same job (i.e. providing vehicle service in a given location).

- ***Loss of subsumption match information*:** As integrated approach provides an overall similarity score based on some similarity distance metric it is impossible to discern from this score whether the match is *plugin* or *subsume* or *vector space neighborhood match* (see chapter 3) [147]. Thus, it may be possible that a high similarity score between two services may actually be the result semantic relatedness between the corresponding feature concepts (such as *vehicle* and *car pickup location*) instead of semantic subsumption (such as *vehicle* and *car*). Because of the lack of distinction between the different types of matches two services having the *O car* and *bus* can be put under the same non-divisible (or atomic) cluster even though at a finer granularity level they should belong to separate clusters.
- ***Effect of the assumption of feature dependency***: Integrated approaches tacitly assume that the features are mutually coupled. This belief arises from the fact that most research approaches have formulated user queries as *tasks* (see chapter 5 for detailed discussion). Due to this reason it is quite obvious to assume a one-to-one mapping between any particular atomic task and a particular type of service. As per one-one mapping, for a query match both given *Input* and desired *Output* components of an atomic task should be satisfied by the same service (or services that are functionally similar). However, the problem of service discovery is not independent from the problem of service composition. This creates a more plausible scenario where a set of "*end*" services is responsible for satisfying the desired *Output* of a query while another set of "*source*" services can take in the given *Input* of the same query. The problem is to select services in such a manner such that there exists at least one valid composition path from a source

service to an end service. From this analysis we learn that integrated clustering of services may actually lead to a poor recall with respect to query matching if we do not distinguish between source services, intermediate services, and end services.

- ***Loss of Semantic Taxonomic Relation***: Integrated clustering approaches take two directions: (i) partitional clustering (such as k-means) and (ii) hierarchical clustering (such as agglomerative). In the first case there is no way that we can define the granularities over the cluster space. In the second case the granularity is defined by the distance measure and the chosen distance threshold on that measure. For any pair-wise similarity computation between a service and an existing cluster at a particular hierarchical level it may happen that, because of the above specified three disadvantages of integrated approach, the hierarchy may not be a semantic taxonomy in the sense that a higher level cluster does not semantically subsume a lower level cluster. In other words, in integrated approach a lower level cluster is a subset (and not a semantic subclass) of the higher level cluster from which it is identified. We cannot define any subsumptive relation between either the members of a cluster or the members of two hierarchically differentiated clusters. This results in sub optimal cluster quality in terms of precision with respect to query matching as there is a probability of false positives within a cluster. Also the lack of semantic taxonomic organization substantially increases query matching computation because: (a) it is difficult to identify the optimal hierarchical level of the cluster space where the average precision and F-score can be maximized, and (ii) for a complete query match we require an exhaustive search within each cluster as a single

match with the mean of a cluster (considering it to be the identity of the cluster) does not ensure that the entire cluster can be considered as a solution to the query.

4.4 Problem Statement - Reformulated

In this section we formalize the problem of service clustering in the following manner: *Given a dynamic set of services* $S = \{s \mid s = \langle I, O\rangle \ni I = \{p \mid p \in \Delta\} \,\&\, O = \{q \mid q \in \Delta\} \,\&\, (\forall q;\, \nexists p \ni q \equiv p)\}$ *generate a set of "feature similar" clusters* $CS = \{C_i \mid C_i \subseteq S \,\&\, \forall s \in C_i\,;\, s_i \overset{IO}{\equiv} s_j;\, i \neq j\}$.

The underlying assumptions of the problem definition are:

- There exists a countable domain collection *D*
- *D* is completely identified and structured into ontologies (Δ)
- *D* is not *covered* (i.e. possibility for addition of new domain ontologies or domain concepts is always open)

In the above formalism there are several observations that are noteworthy:

- All input and output parameters belong to an ontology Δ. In other words they are semantically defined within an ontology. Such definition can be either dynamic (see chapter 3) or borrowed from existing ontologies.
- For all input parameters there cannot exist any output parameter that are *semantically equivalent*. In other words, the semantic definition of parameter types has to be unique. This restriction is imposed because for services if an output is equivalent to the input then basically the functionality of the service is invalid since behaviorally the service always remains in the same functional state.

- *Feature Similarity* (denoted as $\overset{IO}{\equiv}$) is a stratified way of matchmaking services (see chapter 3) where services are pair-wise matched according to a single functional feature (either Input (*I*) or Output (*O*)). This way of clustering is in complete contrast to conventional integrated category learning where all the feature set is treated as a service vector for similarity computation over a vector space. It is to be noted that there is no restriction as such to which feature be chosen for clustering (i.e. input or output). The selection of a particular feature will depend on the discovery algorithm (whether centralized or distributed). In chapter 5 it will be seen that for centralized discovery both the features are selected individually for creating separate cluster sets – one for the Output (termed *O-cluster space*) and the other for the Input (termed *I-cluster space*). While in chapter 6 only the *O-cluster space* is created for distributed agent-based discovery.
- The set of services is dynamic which means that new members can be added into the set non-deterministically and existing members can be deleted non-deterministically. We do not model the underlying stochastic birth-death process in our algorithm.

4.5 Semantic Taxonomical Clustering (STC): Conceptual Foundation

In this section we propose *Semantic Taxonomical Clustering* – an alternative novel algorithm for service category learning. Before discussing the algorithm in details and its advantages over other approaches we first need to lay down the essential conceptual foundation. In our approach we define a cluster space as a set of *service taxonomies*. We first formally define a service taxonomy as follows:

Definition 4.1: A g-type *service taxonomy* (denoted as T^g) is a partial-order $\langle s, \overset{g}{\supset} \rangle$ where s is a service and the order is the *g-relation* $\overset{g}{\supset}$ (see chapter 3) where $g = \{I, O\}$ s.t. there exists a unique supremum (or *least specific predecessor*) called the *root service*. ■

g-type service taxonomy (in brief *g-taxonomy*) T^g has some basic properties as discussed below:

- A taxonomy is a cluster of *feature similar* services where the feature set $g = \{I, O\}$
- *Feature similarity* in a service taxonomy can be of 4 types: (i) *exact*, (ii) *plugin*, (iii) *subsume*, and (iv) *sibling*. For detailed discussion on each of these 4 matches refer to chapter 3.
- A taxonomy is a stratified cluster of feature similar services. This is because the *g-subsumption* relation is either with respect to *Input* (*I*) or *Output* (*O*).
- *Feature similarity* with respect to a taxonomy is non-distance based. In other words, the similarity condition is not based on any measure but rather type of semantic subsumption match (*exact/plugin/subsume/sibling*).

We now define a taxonomical cluster space as follows:

Definition 4.2: A *g-type taxonomical cluster space* (denoted as *CS-*T^g) w.r.t to a particular functional feature *g* is a dynamic set of *g-type taxonomic clusters*. ■

We now define a taxonomical cluster space as follows:

g-type Taxonomical cluster space (in brief *g-cluster space*; figure 19) has several properties that makes it unique from cluster spaces generated in conventional learning algorithms. They are discussed as follows:

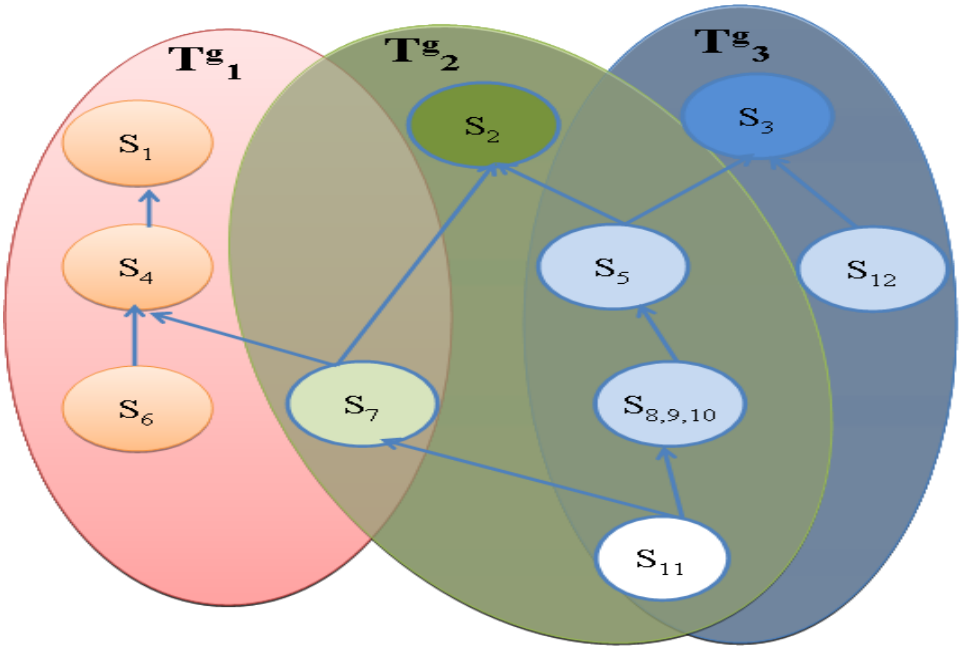


Figure 19: g-Taxonomical Cluster Space

- A taxonomical cluster space is dynamic (i.e. addition and deletion of member services within taxonomies can be non-deterministic). Hence, as we will see in the following proposed algorithm, the cluster space supports online learning.
- There are two types of cluster spaces possible: (i) *O-cluster space* (where the feature similarity is over the *Output* feature) and (ii) *I-cluster space* (where the feature similarity is over the *Input* feature).
- The member taxonomies within the cluster space are not necessarily disjoint from each other. This is so because a particular service can have *g-subsumption* plugin match with more than one service each of which are member of separate taxonomies. For an example, a car rental service having *Output = {car info, rental confirmation}* may have *plugin* matches with both a vehicle rental service having *Output = {vehicle*

confirmation} and a vehicle lookup service having *Output = {vehicle info}* with the *O-cluster space*. In this example the vehicle rental service and the vehicle lookup service belong to two different taxonomies (i.e. taxonomies having two distinct root services).

- The converged topology of the cluster space is independent of the *order of sample selection*. In other words, the temporal order in which services are published into the system does not affect the final cluster space topology (i.e. the number of taxonomies and the partially ordering within each of the taxonomies). This will be evident once we introduce the taxonomical clustering algorithm in the next section.

We now define an *MSP* and an *LSC* of a particular selected service *s* below. These two structures form the basis of the STC algorithm that will be described in the next section.

Definition 4.3: *MSP* (or *Most Specific Parents*) of a given service *s* is a set of services such that: $\forall p \in MSP\ ;\ p \overset{g}{\supset} s \wedge \nexists q \ni p \overset{g}{\supset} q \overset{g}{\supset}$. ■

Definition 4.4: *LSC* (*Least Specific Children*) of a given service *s* is a set of services such that: $\forall m \in LSC\ ;\ s \overset{g}{\supset} m \wedge \nexists n \ni s \overset{g}{\supset} n \overset{g}{\supset} m$. ■

4.6 Semantic Taxonomical Clustering (STC): Algorithm

The basic outline of the proposed *STC* learning algorithm involves "*semantically inserting*" a randomly selected service from a dynamic service set into one or more *g-taxonomies* within the corresponding *g-cluster space*. The insertion of random service is done by searching for the *most specific parents* (*MSP*) and the *least specific children (LSC)* of the service (figure 20). The algorithm utilizes an important property of a *g-taxonomy* to

improve the clustering efficiency. The property has been given in the form of a theorem below:

Theorem 4.2: If for a selected service *s* there exists a non-empty MSP and if there exists a non-empty LSC of *s* then $\forall p \in LSC ; \exists m \in MSP \ni m \overset{g}{\supset} p$.

Proof: As the selected service $s \overset{g}{\supset} p$ hence, and as the MSP exists hence, there must exist *m* such that $m \overset{g}{\supset} s \overset{g}{\supset} p$ ■ .

The implication of the above theorem is that for semantically inserting a selected service into a taxonomy we need to identify the MSP of the service. Once that is done then we can restrict the search for LSC of the selected service to the LSC of each member service within the MSP. This significantly reduces the search space for finding the correct taxonomic position of the selected service. If the MSP of the selected service is empty and the selected service does not have *sibling match* with any of the existing *root services* then the selected service becomes a *root service*. In that case the LSC of the selected service has to be identified from the entire existing cluster space. Otherwise, if the selected service has a *sibling match* then a new *abstract service* is created that subsumes the sibling services. This operation is very significant in the process of service discovery as will be explained later in chapters 5 and 6. Another implication of the above theorem is that the member services in the MSP may not belong to the same taxonomy. In other words, there may exist more than one root services s_r such that $s_r \overset{g}{\supset} m; m \in MSP$. Thus, the selected service may belong to multiple taxonomies (figure 20).

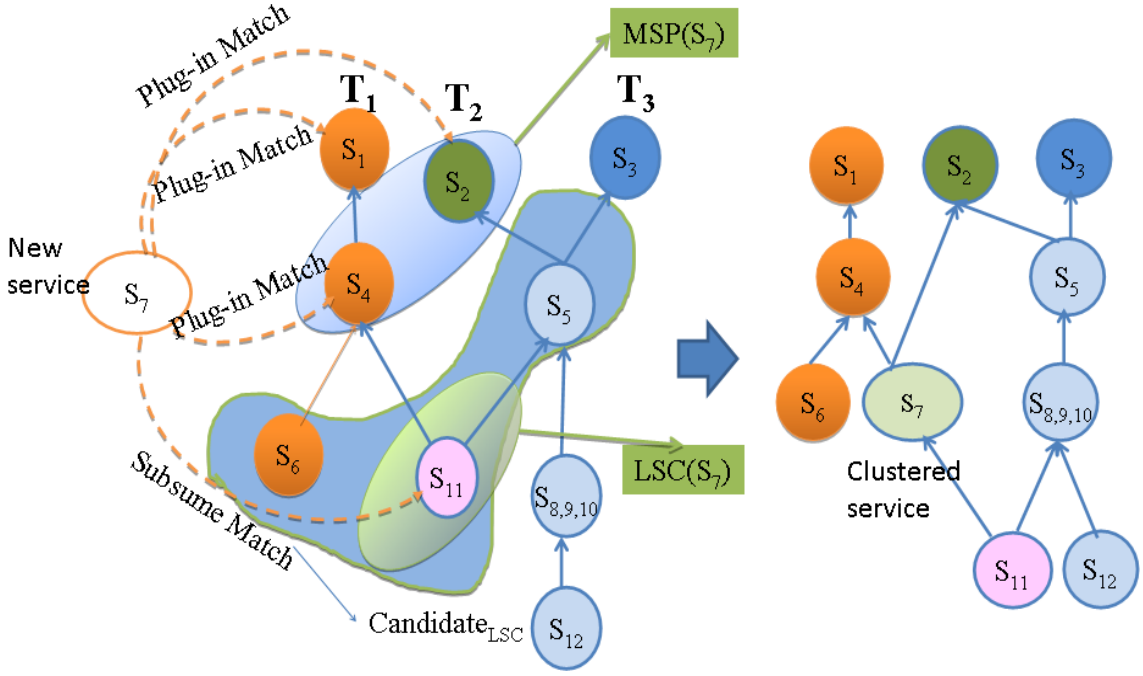


Figure 20: STC Algorithm Outline

The *STC* algorithm returns an instantiated CS when given the dynamic service set *S*. This main algorithm requires two functions: (a) *findMSP* for computing the MSP of a particular service, and (b) *findLSC* for computing the LSC of a particular service. For pair-wise service matching the *g-subsumption* algorithm is used (see chapter 3). It returns 0 if there is no match, 1 if the first argument service subsumes the second argument service (*subsume match*), and 2 if the argument services are *sibling match* under a common abstract parent service. We now provide a detailed outline of the *STC* algorithm along with the *findMSP* and *findLSC* functions that are called within *STC* as follows:

ALGORITHM: ***Semantic Taxonomical Clustering (STC)***
INPUT: sample space $S = \{s_1, s_2, s_3 \ldots\ldots s_n\}$
OUTPUT: cluster space $CS_{1..n}$

```
START
CS = NULL // initially CS is set as empty
for count = 1 to n
{
        sample = randomSelect(S);
        S = S – (sample);
        MSP = findMSP(sample, CS);
        for 1 to MSP.size
        {
                PLSC = PLSC ∪ findLSC(memberOf(MSP)); // PLSC: Potential LSC
        }
        findLSC(sample, PLSC, CS);
}
return CS;
END
```

Figure 21: Semantic Taxonomical Clustering (STC) Algorithm

```
SUB-ROUTINE: findMSP
INPUT: sample, CS
OUTPUT: MSP of sample

START
sample.visited = false;
for count 1 to CS.size
{
        node = extractMember(CS);
        // CASE A: When CS is empty (initial state)
        if node == NULL
        {
                CS = {sample};
        }
        else if(node != NULL && node.visited = false)
        {
                // CASE B: When sample has no parents
                if (g-SubsumptionMatch(sample, node) == 0)
                /* 0 denotes the disjoint relation with  respect to the 'g' dimension*/
                {
                        MSP = NULL;
                        return MSP;
                }
                // CASE C: When sample gets at least one parent
                else if g-SubsumptionMatch(sample, node) == 1)
                /* 1 denotes parent-child relation with respect to the 'g'dimension */
                {
                        node.visited = true; // this node won't be selected again
                        if(node.childrensize != 0)
                        {
                                for count = 1 to node.childrensize
                                {
                                        node = node.child[count];
                                        findMSP(sample, node);
                                }
                        }
                        else
                        {
                                sample.parent[parentsize + 1] = node;
                                sample.parentsize ++;
                                node.child[childrensize+ 1] = sample;
                                node.childrensize ++;
                        }
                }
                for count = 1 to sample.parentsize
                {
                        MSP = MSP ∪ sample.parent[count];
                }
                return MSP;
        }
        else if(node != NULL && node.visited == true)
        {
                continue;
        }
}
```

Figure 22: *findMSP* Sub Procedure

```
SUB-ROUNTINE: findLSC
INPUT: sample, member of sample MSP, CS
OUTPUT: LSC of sample for the member of sample MSP
START
// CASE A: When sample has a non-empty MSP
if member != NULL
{
        for count = 1 to member.childrensize
        {
                if (g-SubsumptionMatch(member.child[count], sample) == 1)
                {
                        LSC = LSC ∪ member.child[count];
                }
        }
        insertMember(CS, sample);
        return LSC;
        }
// CASE B: When sample has an empty MSP
else
{
        for count = 1 to CS.size
        {
                node = extractMember(CS);
                if node.visited == false
                {
                        node.visited = true;
                        if(g-SubsumptionMatch(node, sample) == 1)
                        {
                                LSC = LSC ∪ node;
                        }
                        else continue;
                }
                else continue;
        }
        insertMember(CS, sample);
        return LSC;
}
END
```

Figure 23: *findLSC* Sub Procedure

4.7 Online Learning: STC vs EASY [149]

In an online learning framework we do not have a fixed training dataset (as opposed to supervised learning mode) or a fixed sample space over which unsupervised learning can be performed. Instead services are observed by the learner one at a time without any prior knowledge or estimation of the stochastic process that might govern the occurrences of the services. Thus, it is very difficult to create a statistical training data set that does not suffer

the risk of under-fitting (by excluding significant parameters that have not appeared yet) and over-fitting (by including seen parameters that in the long run prove to be not significant) in the learning process. This is more so because the service categories cannot be estimated clearly since so many categories are evolved online and so many categories can become extinct. Also, unsupervised learning methods that are ad-hoc cannot fit into this framework as well due to the dynamic addition (and deletion) of services.

It is in this kind of framework that STC fits in very well. This is because STC does not require a prior training data. It also does not require the sample space to be fixed during clustering. This is because according to STC newly observed services are semantically positioned within a set of taxonomies within the cluster space after a certain number of *g-subsumption* comparisons. As the algorithm does not involve ad-hoc comparisons within a fixed set of services hence, once a newly observed service is positioned it does not require re-positioning when newer services are observed. However, because of the positioning of these newer services, the relative position of the already positioned services may change automatically. In other words, newly observed services are self-organized by the learner.

In our knowledge there have been very few service category learning algorithms that support such online learning. A relatively recent work by Mokhtar et al [149] has proposed an online category learning algorithm, called *EASY*, that attempts to organize services into taxonomies (called *capability graphs*). However, there are some major drawbacks in their algorithm that makes it not sound and complete. Here we conclude this section by providing a comparative analysis of *STC* and *EASY* as below:

- In *EASY* the semantic similarity measure that has been used for matching is integrated. This introduces the problems of integrated distance-based measures as discussed earlier. Since *STC* is stratified such problems can be avoided.
- In *EASY* the semantic similarity measure is based on Paolucci match order. In chapter 5 we have shown that there are some serious flaws in the match order. Thus, *EASY* suffers from serious accuracy problem.
- In *EASY* a separate comparison process is carried out from the bottom of the taxonomy after a set of Most Specific Parents are found. This step is not necessary as per the theorem in the previous section. STC, on the other hand, uses the property shown in this theorem and hence, avoids redundant computations.
- According to *EASY* a newly selected service must have a predecessor if it is subsumed by a root service within a taxonomy. However, this is not always true since the new service may become a sibling of a set of services under the common predecessor – the root service. As the sibling case is completely ignored by *EASY* (unlike *STC*) hence, the learning algorithm may not terminate at all under those situations.
- It can happen that a newly selected service is neither subsumed by nor subsumes any of the root services. However, it can be a sibling with one or more root services under a common *abstract service* (as discussed earlier). Since this case is ignored by *EASY* (unlike *STC*) hence, we see that there can be false negation during clustering.

In *EASY* services are organized into taxonomies called *capability graphs*. *Capability graphs* are unique in the sense that each graph is indexed by the ontologies that contains the semantic descriptions of the member services within the graph. Thus, if a service uses

parameters from the domains *vehicle* and *user profile* then it will be inserted into a capability graph that has services from these two domains only. However, the drawback in this approach is that since, there may not be a single existing capability graph that is indexed by both the two domain ontologies hence, the newly selected service may actually be inserted into two separate *capability graphs* (each indexed separately by *vehicle* and *user profile* respectively). This causes redundancy and increases the number of comparisons needed for further service categorization as well as query mapping (will be discussed in chapter 6).

4.8 STC Analysis: Soundness & Completeness

We now provide the mathematical proofs for the soundness and completeness of *STC* as follows:

Theorem 4.3: *STC* is sound with respect to any arbitrary sample space S.

Proof*:* For any arbitrary sample space S if an arbitrary sample *s* is selected then it has to either match with one or more of the existing taxonomies (say T_E) or none. All possible answers in the algorithm can be represented in the form: $N_i \overset{g}{\supset} s \overset{g}{\supset} N_j \; where \; N_i \subseteq \{\varepsilon, n_{1i}, n_{2i}, \dots, n_{mi}\}$; $N_j \subseteq \{\varepsilon, n_{1j}, n_{2j}, \dots, n_{kj}\}$ such that $n_i, n_j \in T_E$. There cannot be any answer beyond this bound (*s* has to be parent of some nodes and child of some nodes where these set of nodes can be empty (i.e. ε)). To prove the soundness we need to prove that *N* is always correct in the algorithm. If *s* is a parent of some n_{pj} then for each of the siblings of n_{pj} a new search is conducted; thus, in the worst case, traversing the entire cluster space. Hence, N_j includes all *n*s that are children of *s* and cannot include any *n* that is not (in that case it will be included in N_i if it is a parent). Therefore, N_j is always correct. Similarly, if *s* is a

child of some n_{qi} then for each of the siblings of n_{qi} a new search is conducted; in the worst case traversing the entire cluster space. Hence, N_i includes all *n*s that are parents of *s* and cannot include any *n* that is not (in that case it will be included in N_j if it is a child). Thus, we see that there is no other *N* apart from N_i and N_j. Therefore, *N* is a correct and tight bound. Thus, the algorithm is sound. ■

Theorem 4.4: *STC* is complete with respect to any arbitrary sample space S.

Proof*:* The algorithm is complete because for any arbitrary sample *s* the algorithm gives an answer in the bound $N_i \overset{g}{\supset} s \overset{g}{\supset} N_j \; where \; N_i \subseteq \{\varepsilon, n_{1i}, n_{2i}, \dots, n_{mi}\};\; N_j \subseteq \{\varepsilon, n_{1j}, n_{2j}, \dots, n_{kj}\}$. As the answer is always correct and tight hence, *s* cannot be a false negative. ■

4.9 Results

We tested the runtime performance of the proposed *STC* algorithm on a system having a CPU cycle of 1.4 GHz and a RAM of 2 GB. The performances are measured based on: (a) randomly generated synthetic services and (b) OWL-S TC v.2 test set of 871 web services collected from different web service registries. The clustering performance is evaluated on the basis of: (i) average runtime for clustering under an online learning framework, (ii) average number of sample hit count (i.e. the number of *g-subsumption* comparisons) during clustering, and (iii) effect of stratification in the proposed *STC* algorithm (as compared to a non-stratified nearest neighbor based online learning algorithm). As the learning framework is online hence, for both the synthetic dataset as well as OWL-S TC dataset services were drawn from the set in random temporal sequence.

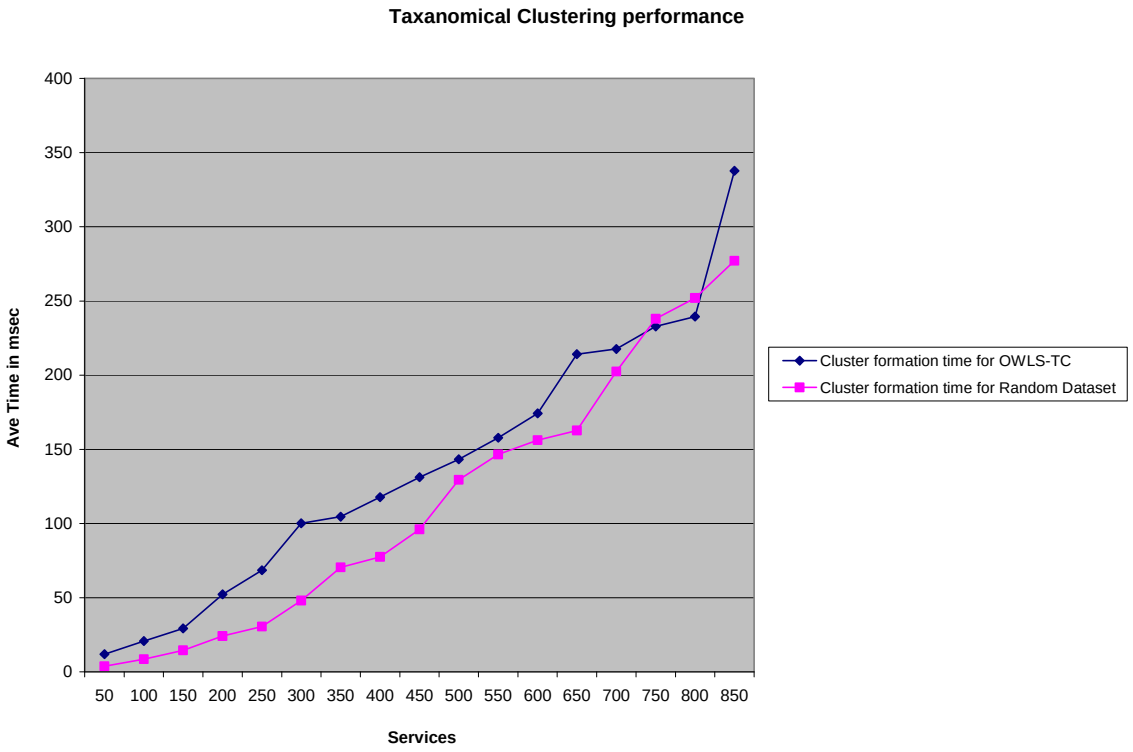


Figure 24: Average Runtime Performance of STC

Random Service Generation: For generating the random sample space we designed a simulation platform where sample spaces of different sizes (50 to 850 web services) were randomly generated using a domain space that consisted of 10 domain ontologies (with an average number of concepts set to 300). An average parameter size of 5 was set for the simulation. Service parameters where chosen randomly from the 10 domain ontologies such that the *Input* feature of each service is distinct from the *Output* feature.

Average Clustering Runtime: When we conducted our experiment with synthetic simulated service sets we found a fairly good runtime clustering performance within an average range of 0.003 seconds (for 50 samples) to 0.277 seconds (for 850 samples) (figure

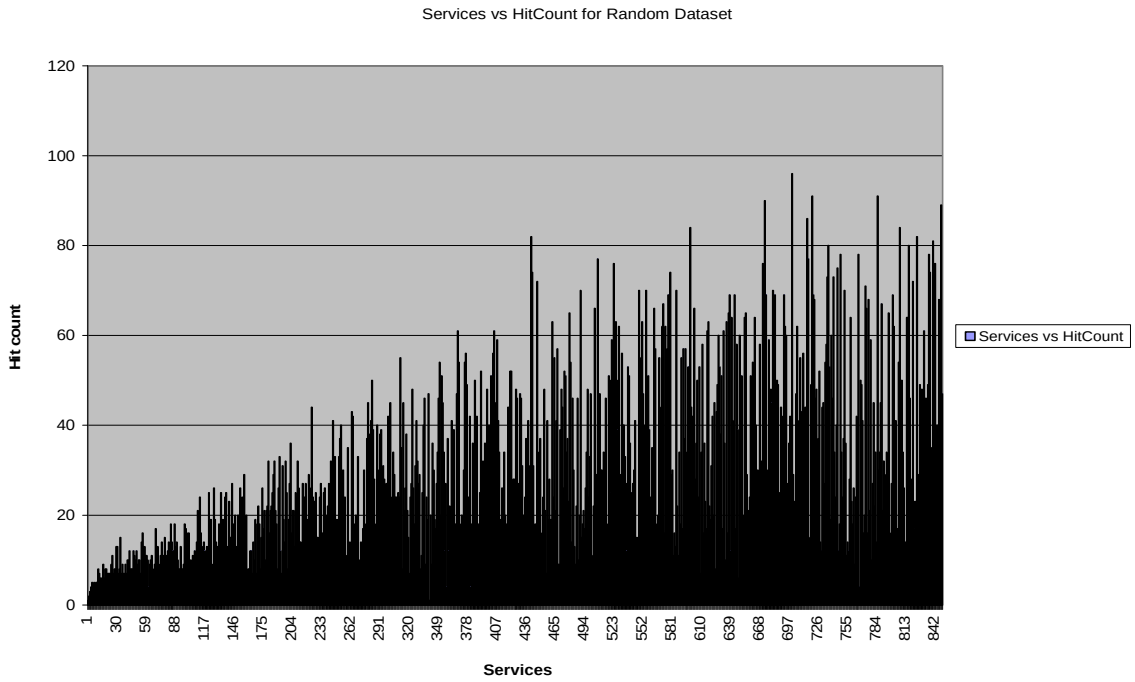


Figure 25: Average Number of Hit Count over Random Sample Space

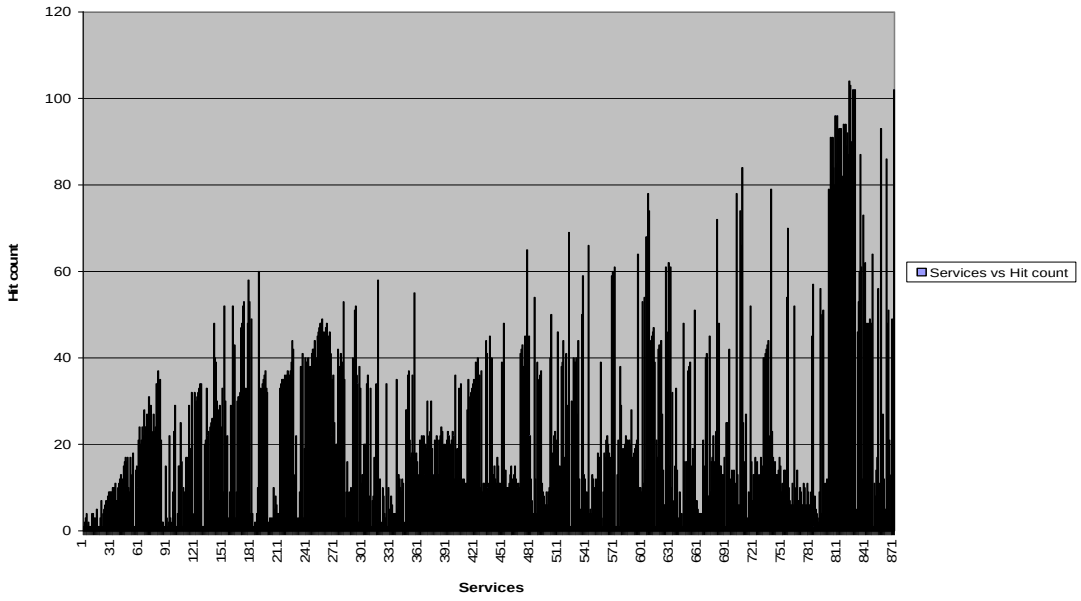


Figure 26: Average Number of Hit Count over OWL-S TC

24). For the OWL-S TC set the runtime was approximately similar with an average range of 0.011 seconds (for 50 samples) to 0.337 seconds (for 871 samples).

- ***Average Hit Count***: Hit count is basically the number of *g-subsumption* comparisons a newly observed service has to go through before it can finally be inserted into its correct position within the currently existing cluster space. This can be another way of understanding the clustering performance. If the average hit count for clustering a set of services is low then it means that it is easier computationally to insert a newly observed service into the cluster space. On the other hand if the hit count is high then the cluster space topology is not favorable for the new service and hence, the computation cost increases. We saw that for random sample space the highest number of hit count is 96. The mean hit count was 29.1 (figure 25). From this observation we can conclude that the clustering algorithm is justified in the worst case when a maximum number of only 96 hits is recorded for placing a particular random sample within a cluster space of 850 services. For OWL-S TC sample space we recorded a highest number of 104 hits for a space of 871 services while the mean hit is 23.4 hits (figure 26). Both the analysis shows that approximately only 3% of the entire cluster space has to be hit before a sample can be clustered. Thus, although the worst case complexity for STC is quadratic still the amortized complexity is very promising.
- ***Effect of Stratified Clustering*****:** We wanted to observe the runtime performance improvement of the stratified clustering approach as compared to an integrated distance-based clustering approach. For that we chose the *SGPS-based semantic distance measure* proposed by us in [189]. The learning algorithm that was implemented in this

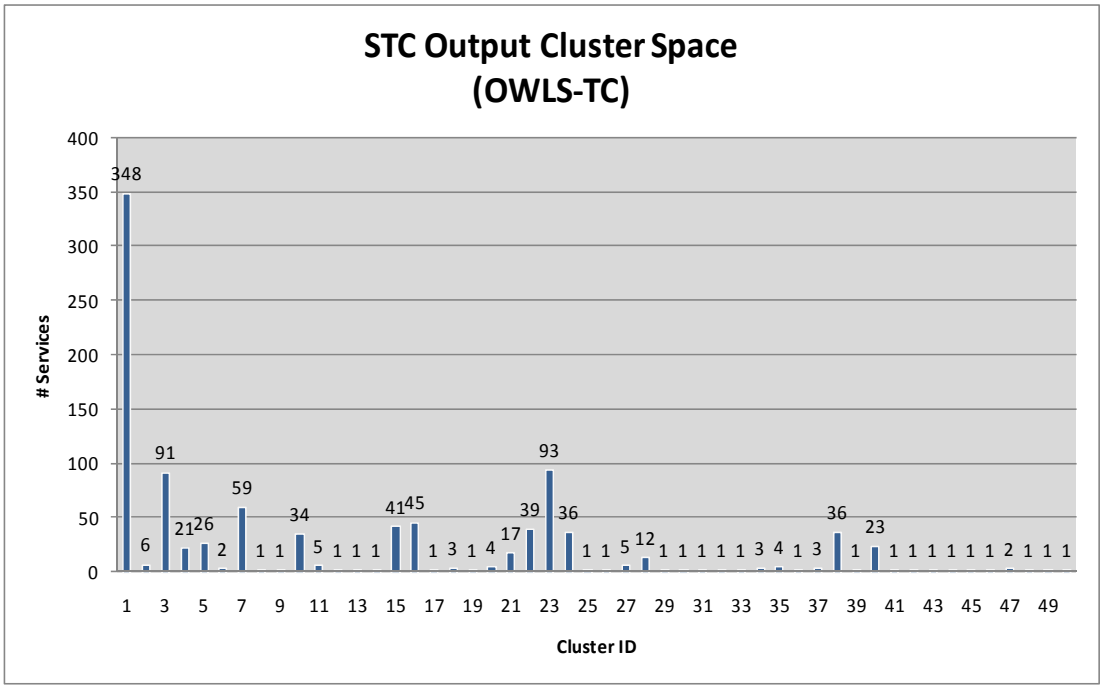


Figure 27: Output Cluster Space Generated by STC

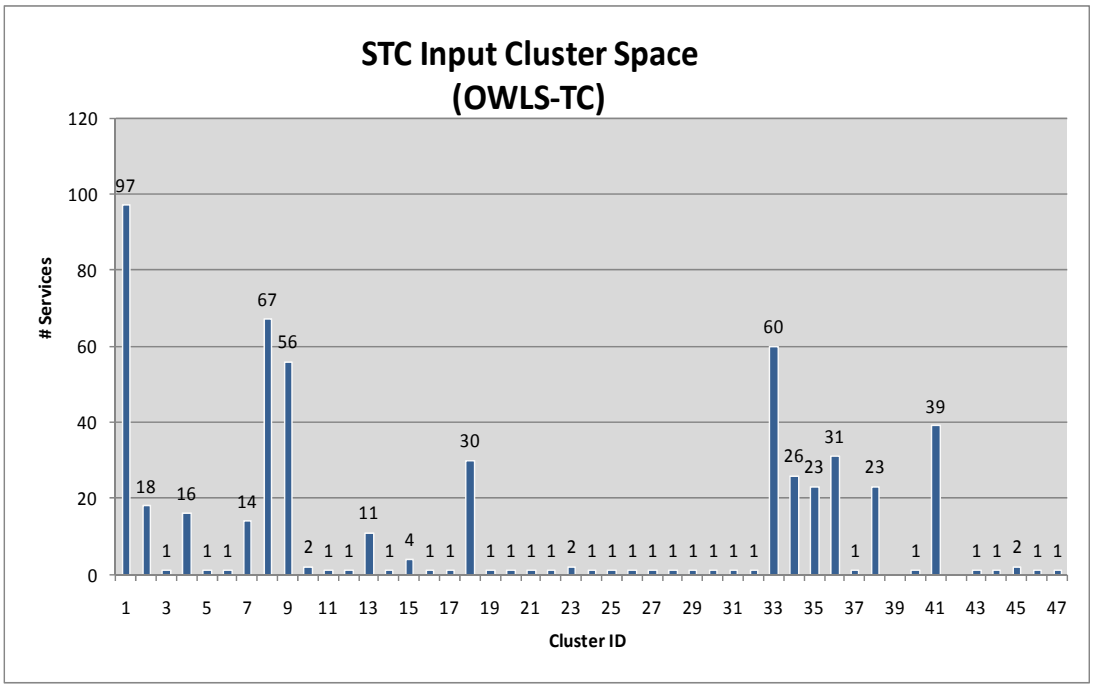


Figure 28: Input Cluster Space Generated by STC

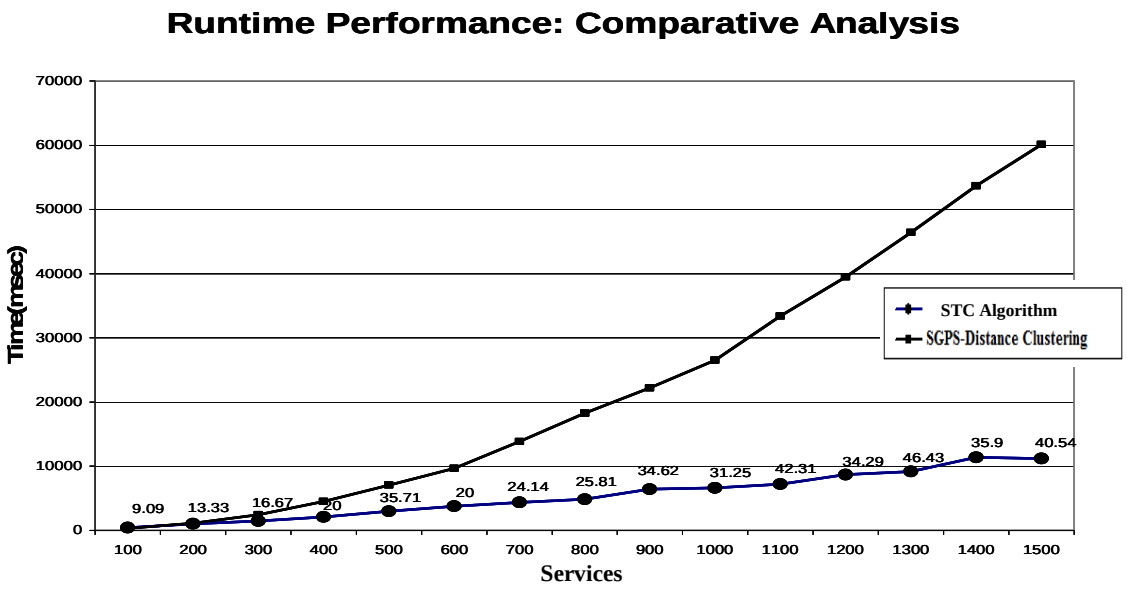


Figure 29: STC Algorithm vs. Integrated SGPS-distance based NN Clustering

work was nearest-neighbor based. For the comparison we used a randomly generated synthetic dataset of 1500 services. We observed a significant improvement in performance as the number of services increases (figure 29). This is because of two major reasons: (i) the pair-wise similarity computation for the integrated *SGPS measure* is significantly higher than that of *g-subsumption* comparison (see chapter 3 for more details) and (ii) the amortized number of pair-wise *g-subsumption* comparisons needed for *STC algorithm* is significantly lower than the amortized number of pair-wise comparisons needed for the nearest-neighbor based algorithm.

For evaluating the accuracy of the proposed STC algorithm we used the standard web service test dataset OWL-S TC v2. The first objective of our experiment was to understand how close STC fits the given service descriptions as compared to the given categorization of

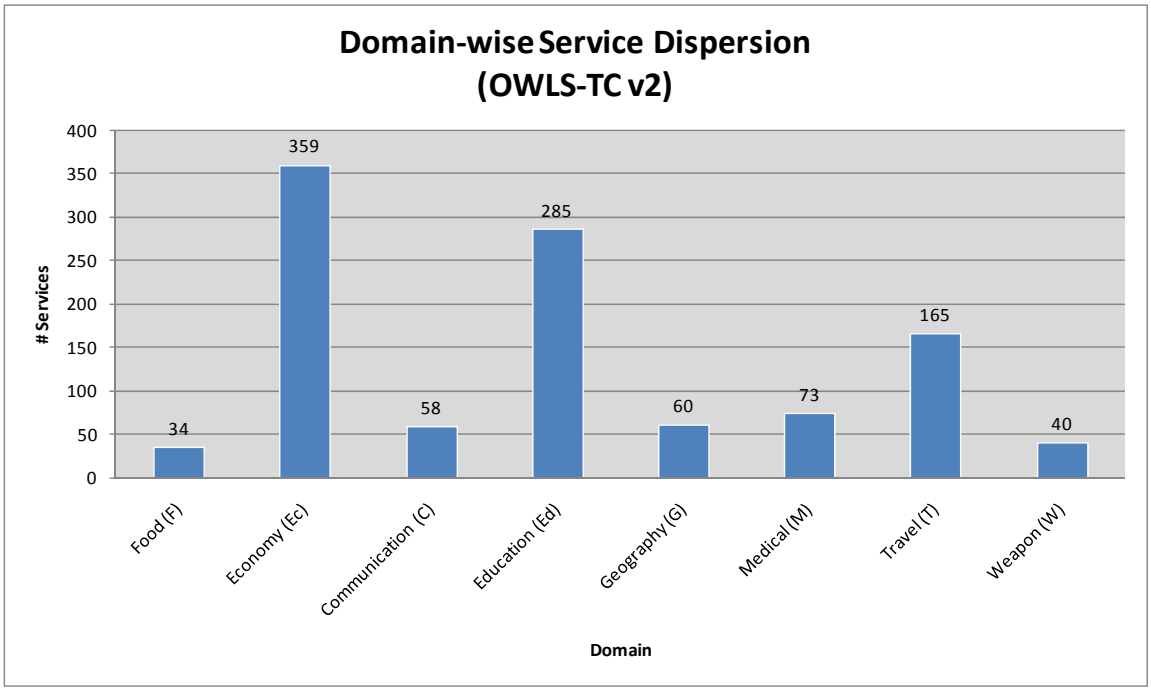


Figure 30: Distribution of OWLS-TC v2 Web Services According to 8 Domains

these service descriptions in terms of their corresponding domains. In order to meet the objective we first define two important measures:

Definition 4.5: *Domain-Precision* with respect to a given service category C and a given service domain D (denoted as *Pr(C, D)*) is defined as the ratio of the number of services in D that are categorized as members in C (say $N_{C,D}$) by any given learning algorithm L vs. the number of services categorized in C (say N_C) by L. Numerically this means: $Pr(C, D) = N_{C,D} / N_C$. ■

Definition 4.6: *Domain-Recall* with respect to a given service category C and a given service domain D (denoted as *Re(C,D)*) is defined as the ratio of the number of services in D that are categorized as members in C (say $N_{C,D}$) by any given learning

algorithm L vs. the number of truly correct services in D (say N_D). Numerically this means: $Re(C, D) = N_{C,D} / N_D$. ■

Note that the above definitions give us a way of computing precision and recall of a given learning algorithm that is completely independent of any query (as opposed to the more conventional query based precision/recall computation which we will discuss later in this section). However, the definition is still a subjective evaluation as it requires human judgment for estimating N_D (number of truly correct services in domain D).

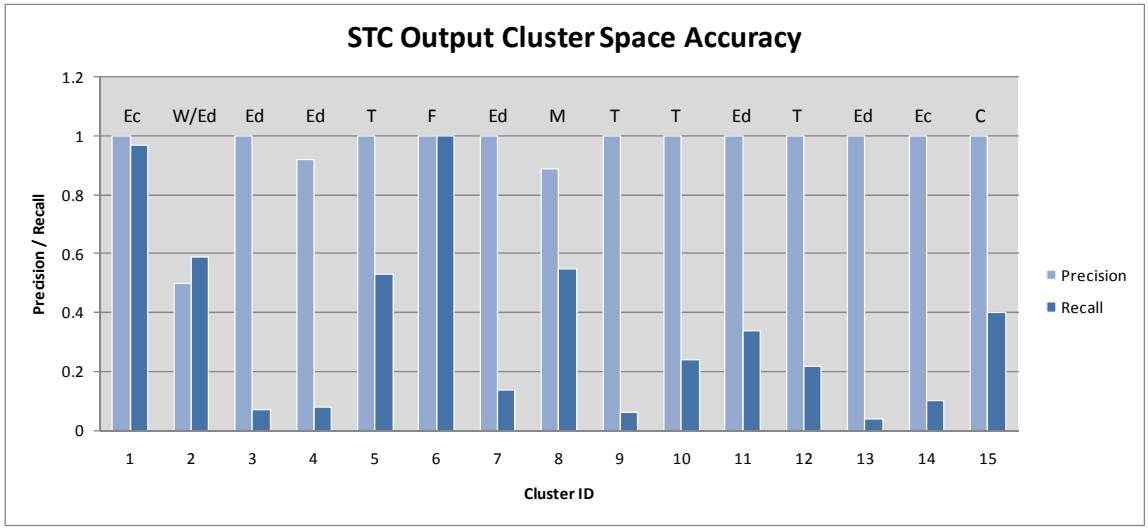


Figure 31: Domain-Accuracy of Output-cluster space Generated by STC

While evaluating the *domain-precision* and *domain-recall* of STC we took the standardized human evaluation included within the OWLS-TC v2 dataset. In the OWLS-TC dataset there are 8 web service domains as shown in figure 30. The innate assumption implied within the expert evaluated classification is that all domains are mutually disjoint. We first observed the accuracy over the *O-cluster space* of STC (figure 31). We observe that there are 50 clusters that are generated (figure 27). However, out of these 50 clusters there

are only 15 clusters that are significant in terms of number of services per cluster. In figure 31 we make a comparative analysis of the *average domain-precision* (in light blue) and *average domain-recall* (in deep blue) of each of these 15 clusters when compared to the domain-wise classified web services in the OWLS-TC dataset. The *average domain-precision* in this context is computed as: $\mathrm{Pr}(C_i^g) = \frac{\sum_{D=1}^{8} \mathrm{Pr}(C_i^g, D)}{8}$. Similarly the *average domain-recall* is also computed as: $\mathrm{Re}(C_i^g) = \frac{\sum_{D=1}^{8} \mathrm{Re}(C_i^g, D)}{8}$. We observe that the average precision for almost all the significant clusters (except for one) is close to 1 while the average recall is comparatively low in most cases. Upon analysis we understand that the O-cluster space is strong enough to represent each of the domains in OWLS-TC that supports our argument that the *Output* parameter is the most significant service feature in understanding service functionality. Hence, STC was able to reduce false positives within its *Output* taxonomies (clusters) by restricting inclusion of services to only those that have mutual *O-subsumption* matches. A very interesting observation can be made with respect to this result: most of the domains have been split over the cluster space. For an example, the domain *Economy* has been split into 2 clusters each having average precision 1 while one having the recall significantly higher than the other. This phenomenon occurs because STC does not assume clusters to be disjoint. Hence, there may be two different functionalities that describe only services within the *Economy* domain in OWLS-TC. Each functionality represents a separate (although overlapped) *O-taxonomy* (i.e. cluster). When we compare this result to the *I-cluster space* generated by STC over the same dataset we find that there

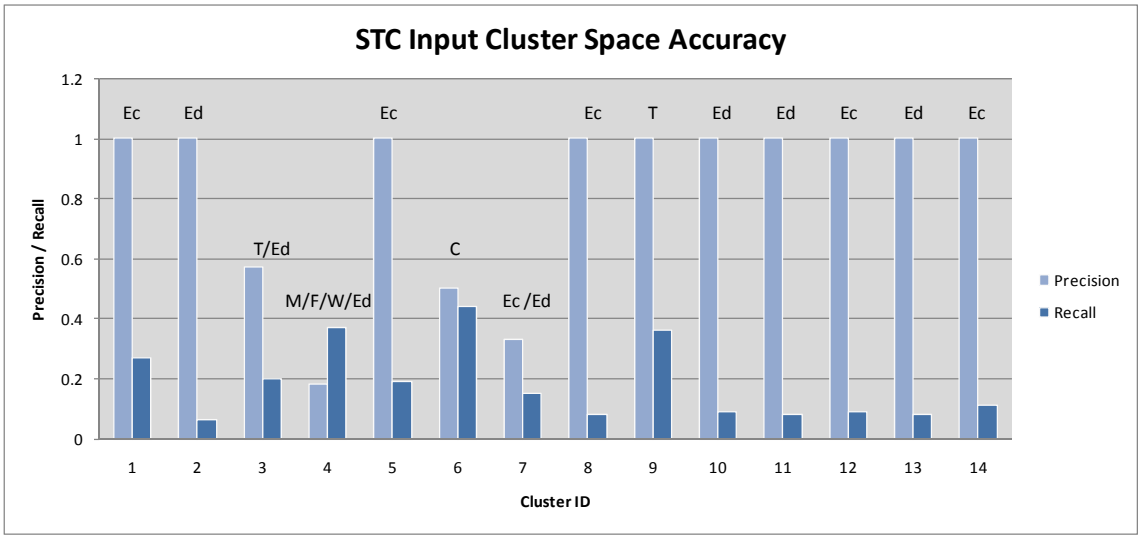


Figure 32: Domain-Accuracy of Input-cluster space Generated by STC

are 45 clusters (figure 28) out of which only 14 clusters are significant. On observation of accuracy in terms of *domain-precision* and *domain-recall* we saw that overall precision and recall dropped (figure 32). Another very interesting observation can be made within the *I-cluster space*: more number of clusters are "*mixed bag*" in nature in the sense that more than one domains are significantly represented (in terms of precision and recall) by each of these *mixed bag* clusters. This phenomenon occurs because the *Input* feature of a service does not adequately represent its functionality. Hence, there can be several services representing different domains that may have the semantically similar type of input parameters (i.e. they are mutually *g-subsumption* matches where $g = I$). For an example the 4th cluster in the figure equally represents 4 domains: *Medical*, *Food*, *Weapon*, and *Education*. The overall *domain-recall* for both the *O-cluster space* and the *I-cluster space* is relatively low because the OWLS-TC expert classification of services into domains are primarily based on service functionality but more on the thematic category of the services. In other words, there may be

services that have quite different functionality although they can pertain to the same domain thematically. For an example, *car price-lookup service* and *car rental service* generate different output (i.e. functionality) while they belong to the same domain *Economy* (i.e. services related to price).

4.10. Conclusion

There are several advantages of the proposed STC learning algorithm as compared to conventional service category algorithms. In this section we discuss each of them as follows:

- ***Eliminates Assumption of Disjoint Cluster***: Since a service can be subsumed (*g-relation = subsume*) by more than one services belonging to different taxonomies (i.e. taxonomies having distinct *root services*) hence, the problem with the assumption of disjoint cluster (as discussed in earlier section) has been eliminated in the proposed STC algorithm.
- ***Eliminates Centroid Problem***: As the algorithm does not require the prior estimation of the number of clusters (i.e. taxonomies) and is not centroid based hence, the problems that are innate in partitional clustering algorithms are no longer relevant in STC.
- ***Supports Online Clustering***: As discussed earlier STC fully supports online service category learning.
- ***Eliminates Problems of Similarity Measure***: Since STC is not based on any similarity measure hence, the problems of distance-based learning algorithms can be easily overcome. Also, since the approach is stratified hence, the problem of integrated similarity measure can be resolved as well. Further elaboration of how the stratified approach helps in service discovery will be done in chapter 5 and 6.

- ***Computationally Efficient***: The time complexity of STC is $O(n^2)$ in the worst case since there can be n(n+1)/2 *g-subsumption* comparisons at the most. Each *g-subsumption* operation between two services takes constant time (discussed in chapter 3). However, we observed under empirical study that the amortized number of *g-subsumption* comparisons is ~ 3% of the services existing within the cluster space in an online learning framework (for more discussion see next section).
- ***Accuracy Theoretically Sound and Complete***: According to the theorems that prove the completeness and soundness of STC we can say that "theoretically" the proposed algorithm has an F-score of 1 (i.e. 100% precision and 100% recall). However, in reality, this totally depends upon the assumption that the semantic descriptions of every service are valid, correct, and complete. In other words, the service providers should not make gross mistake while semantically defining the functional parameters of their published services.

CHAPTER 5

ALNet: EVENT-DRIVEN PLATFORM FOR SERVICE COMPOSITION

5.1 Introduction

The previous two chapters covered the following foundational requirements of service discovery and composition:

- *g-subsumption*: Efficient service matchmaking for improved service organization and query matching .
- *Desire-based Query Modeling* (*DQM*): Efficient query modeling for improved service discovery and service composition.
- *Semantic Taxonomic Clustering* (*STC*): Efficient service organization for improved service discovery.

In this chapter we are going to propose a novel asynchronous event-driven platform called *Activity Logic Network (ALNet)* for service discovery (that has been outlined in the previous chapter) and service composition. The proposed approach draws inspiration from the more recent Event-driven SOA (ED-SOA) architecture that has been proposed in [11 - 17]. In this way *ALNet* shifts significantly from the more conventional broker dependent *pull-based* SOA paradigm where a consumer application needs to *pull* services from the broker after the discovery process in order to *bind* itself to the discovered services. As mentioned in the introduction chapter, traditional SOA based systems enforce synchronous binds during runtime. This is because services are in general stateless and hence, service output has to be gathered over a synchronous channel. In contrast, systems that are built over

ED-SOA principles model all state transitions within the system as asynchronous *events*. These transitions include services as well as consumer queries. The underlying design model is that of *publish-subscribe-notification* where services are published as events to a middleware while consumers subscribe to their required service types. If there is a match then the middleware notifies the best matching service asynchronously. The main difference lies in the fact that in conventional SOA it is the job of the consumer application to bind synchronously to a matched service while in ED-SOA it is the middleware that asynchronously calls the matched service and then provides the output to all the consumer applications that have subscribed for the service.

ALNet is a specific type of ED-SOA based service discovery and composition platform. However, it is quite different from the usual *publish-subscribe-notify* model in that services are published not to be subscribed to by some consumer applications. Rather the middleware generates a novel service dependency graph, called *Activity Logic Network* (*ALNet*), that denotes the pair-wise functional dependency of the published services. There is no centralized *event-library* that stores a priori all possible event definitions within the system. A service when gets executed generates a *service event*. Consumer requests are also treated as events (called *user event*) on-the-fly in accordance with the *DQM* model proposed in the previous chapter. The *ALNet* services differ from conventional published service descriptions in the sense that each service has a set of semantic interpretations for interpreting different events (*service events* as well as *user events*) based on their observable state changes. These interpretations are represented in the local knowledge bases of the registered services. The knowledge bases are designed according to a novel ontological

framework called *CAOFES*. Based on such interpretations each user event is *handled* on-the-fly locally by the published services (in a proactive way) as well as globally by the middleware. Thus, the *ALNet* framework is adaptable to newly observed undefined events as well. The *event handling* process includes a 2-phase service discovery algorithm, called *ALNetSniffer*, outlined in the previous chapter as a part of the *event-handling* as well. Hence, service discovery, selection and composition are tied together into a common problem – *event-handling*.

The chapter begins with an introduction of some significant research works in the area of centralized service discovery and composition. It then discusses the problem of service composition reformulated as an *event-handling* problem. The discussion includes a new semantic way of interpreting events called *Event Notability Theory*. It also details the corresponding semantic ontological framework called *CAOFES* for facilitating such interpretation. After that the chapter unfolds into a full length proposal of the *ALNet* data structure and the underlying *ALNet* architecture. After laying the architectural foundation the chapter then makes a detailed treatment of the proposed 2-phase service discovery algorithm called *ALNetSniffer* and its relation to *STC* - our proposed service category learning algorithm. This follows with the proposal of the optimal composition discovery algorithm – *ALNetComposer*. Later on the chapter discusses another new concept called *Situation Boundary* (*SB*) that helps to optimize the runtime event-handling performance and then finally proposes the *event-handling* algorithm – *SBTraveller*. The chapter concludes with a detailed query-based accuracy evaluation of *ALNetSniffer*, the query processing overhead by *ALNetSniffer*, and the composition performance of *SBTraveller* algorithm.

5.2 Related Work

In most research works the problem of middleware based service discovery has been treated as a special case of the more generic information retrieval problem. The common principle is to organize (functionally) similar services into categories that are then stored in backend systems. Back-end systems can be classified into two types: (i) centralized index table based (such as UDDI [8], Jini [191], SLP [192], m-SLP [193]), and (b) Distributed Hash Table based (such as CHORD [155]). Consumer requests are then directed to the backend systems as queries. Currently, the most popular backend implementation for service discovery is UDDI. UDDI is organized based on pre-defined categories (as standardized in NAICS [179], UNSPSC [178]). However, the service descriptions within UDDI are not categorized with respect to their functionality. Thus, UDDI cannot support effective content-based query match. Many other researches propose an extension to the existing UDDI structure by adding semantic descriptions of the services. The semantic description can be at three levels: (i) *functional* (such as OWL-S [50]), (ii) *contextual* (CC/PP [194], UWL [195]), and (iii) *QoS* (such as in WSLA [196], WSOL [SCP 197]).

There has been a lot of research on service composition – both static as well as dynamic. A detailed discussion of these works has been provided in chapter 2. Although there is not much research on modeling ED-SOA systems as service dependency network yet a very close approach to *ALNet* can be found in the work by Lang & Su [198]. In that work the authors have proposed a service network called *service dependency graph* (SDG) that is based on AND/OR graphs [199]. The problem defined thereby is to search for a matching structure of *operations* as required by the users. Queries are modeled as a structure

containing user input, desired output and optional sequence of operations. In this way queries are not restricted to tasks only. Composition takes place following a back-tracking (or bottom-up) algorithm starting from the desired output data node. However, although the overall problem objective is same as compared to our proposed *SBTraveller* algorithm yet there are some major differences between the two. A comparative analysis of *ALNet*, SDG, and its variant *SDG+* [200] will be given in the conclusion and discussion section of this chapter.

5.3 Event Handling: Service Composition Problem Reformulated

In the proposed model user requests/interactions (in the *DQM* format; see chapter 3) and services (in the *g-array* format; see chapter 3) are treated as *events*. An event is an activity executed by some agent (i.e. the users and the devices in an SOA system) in order to bring about some changes in the states of the world. The world is comprised of the system and its surrounding environment. However, not all activities can be treated as events. A state change of an activity has to satisfy two conditions so that the activity can qualify as an event: (a) it has to be observable by an agent, and (b) it has to invoke a defined interpretation to an agent. For an SOA system the user's activities are perceived as *user events* if a notable change in the environment state provokes a system of services towards a specific understanding and invokes subsequent responses to the activities as a *handling process*. We term such a handling process as *event-handling.* The services organize themselves on the run based on such interpretations and produce a resultant set of events that in turn has a specific interpretation to the user/s. This resultant set of events is called the *target event set* and is the

desired goal of user agent (i.e. the Q-T1 component according to the *DQM* format). The Q-T2 component of the user event that provoked such a dynamic collaboration is called the *initial event*. User events and services bear causal relations with each other. A particular service can be triggered only if its corresponding causal events (services and/or user-events) are triggered. If a service/s has its cause as the *initial event* then it is called the *source service/*s for that event.

The uniqueness of the proposed *event-handling process* is that it does not require a task-analyzer to form a task-based request. Instead, user requests are perceived by the proposed *ALNet* system as user events. It then responds to these events by triggering the corresponding services registered to it. *Event-handling* has a two-fold objective: (a) to obtain the best *target event set* for an *initial event*, and (b) to minimize the cost of the overall process. Hence, in our model the service composition problem is not primarily a global path optimization problem but rather an end optimization where the best possible *end service* has to be triggered starting the process from a set of source services that has *reachability* to these *end service/s*. The best *end service* is the one that is functionally most similar to the desired *target event*.

There are three basic operations over an ALNet instance: (i) search for end service nodes that can trigger the target event(s), (ii) search for source nodes that have reachability to at least one of the end service nodes, and (iii) select the service nodes starting from a source node so as to achieve the optimal event-handling. The first two operations form the basis of the 2-phase service discovery algorithm, called *ALNetSniffer*, proposed in chapter 3. In order to perform the last operation we need to check the *reachability* between every

selected node (starting from the source node) and the selected end node. *Reachability* checking and subsequent service selection has to be done on the fly. In this paper we have proposed an encoding based technique for performing such dynamic checking in linear time. The process of reachability checking and service selection results in a particular composition path from the source service to the end service. Reachability has been discussed in section 6.5.3. A full length discussion on research work in this problem has been given in chapter 2.

5.3.1 Event Notability Theory

In a given SOA based system activities such as consumer request, consumer interactions for occasional feedbacks into the system, and executed service operations can be treated as events. In the proposed *ALNet* model an *event* is an activity executed by an agent (i.e. the consumers or the devices hosting services) that is observable by the system and has specific interpretations for a set of agents within the same system. *Event Notability Theory* is a formal logic-based novel modeling of pro-active event-based systems. The model treats all processes as *activities* and provides a semantics for *event* that is distinct (although derived) from the semantics of an activity. There are two kinds of activities in a pervasive system: (a) *user agent activity*, and (b) *device agent activity*. An activity is said to be an event if the activity has a specific interpretation to a set of agents. Formal definitions of activity and event will be given in later sections. We think that agent should be the core concept in a pervasive system management framework as the definition of other concepts (such as events, context, etc) is subjectively dependent upon agent. In other words what is event or

context for one agent may not be so for other agents. Hence, we first provide a formal definition of agent.

Definition 5.1: An *agent* (denoted as *Ag*) is an entity that has a well-defined knowledge base (denoted Δ) and there exists an interpretation τ over Δ (denoted Δ^{τ}) such that τ triggers off some role axiom A where $A \in \Delta$.■

The definition implies that: (a) the knowledge base (KB) of the agent has to be well-defined in terms of syntax and semantics, (b) there exists a set of role axioms (denoted as U^R) within the knowledge base where U^R is basically a set of first-order response rules written in the event-causal_condition-action (ECA) format that is commonly used for defining policy roles of agents [11], and (c) the knowledge-base should have an interpretation that triggers off at least one of the response rules. The significance of the third implication is that an agent having a KB without an interpretation is meaningless for a particular world. The agent may be able to perceive entities but cannot understand their meaning (if the entities are not interpretations) or cannot response to the perceptions (if the entities are not interpretations). Thus, the above definition is an axiomatized version of what we loosely mean by agents – viz. humans, devices, humanoids, software agents, etc. A derived implication of the definition is that the agent should have the cognitive capability to perceive (*perceptibility*) and the capability to response (*notability*). We will define these two concepts in later sections. It should be noted that agents also execute the activities apart from perceiving or understanding them. However, the capability of activity execution is not a necessary condition for an agent (there can be agents that are purely *event sinks* [16]). However, in our system there cannot be an agent that has takes action without having

perceptibility and notability (that is purely *event sources* [16] are not allowed in our system). This is because we do not think that an agent producing arbitrary events is of much use for a pervasive system. Such agents cannot be associated with any desire, intention, and belief. We term such agents as *nuisance agents*. Nuisance agents generally add up to the chaotic property of a system by introducing non-deterministic noise and conflicts.

Interpretation of an activity as an *event* follows from the perception of the change in states that is triggered by the activity. A state of an activity is the set of time variant vectors that describe the system. We term the state vector space as *State Vector Space*. *State Vector Space* is a 5-dimensional vector space having five state element vectors: (i) Background ($\overrightarrow{Bg}$), (ii) Object ($\overrightarrow{Ob}$), (iii) Agent ($\overrightarrow{Ag}$), (iv) Spatial ($\overrightarrow{Sp}$), and (v) Temporal ($\overrightarrow{Tp}$). We call such a vector space as *state vector space* (denoted as $\overrightarrow{SV}$). The state vector elements are defined in terms of the activity with which they are associated. Thus, we can describe the state of an activity (and hence, an event) with the help of the state vector space. We explain each of the five elements below:

a) *Background Vector* ($\overrightarrow{Bg}$): Specifies state information of entities around the place of execution of an activity. Example: *A player kicked a football in a stadium where cheer leaders were present* – here *cheer leader* is the background vector.

b) *Object Vector* ($\overrightarrow{Ob}$): Specifies the entity whose state is getting affected by the execution of an activity. The state can be the location or dimension or some other property of the entity. It is to be noted that the entity is not the agent who executes the activity or the

agent who perceives the activity. Example: *A player kicked a football* – here *football* is the object vector.

c) *Agent Vector* ($\overrightarrow{Ag}$): Specifies the active agent executing an activity (i.e. the agent profile, capabilities, location, etc) and also the passive agent who perceives the activity. In the above example *player* is the agent (in this case he is active agent).

d) *Spatial Vector* ($\overrightarrow{Sp}$): The place where an activity occurs (in terms of geography, specific address, relative location with respect to a specific address). Example: *A player kicked a football in a stadium* – here *stadium* is the spatial vector (in this case relative location).

e) *Temporal Vector* ($\overrightarrow{Tp}$): The time when an activity occurs (in terms of day, year, month, morning, afternoon, evening and night). Example: *A player kicked a football during a morning match* – here *morning* is the temporal vector.

We now formally define *State Vector Space* as follows:

Definition 5.2: A *state vector space* (denoted as $\overrightarrow{SV}$) is a vector tuple $\langle \overrightarrow{Bg_v}, \overrightarrow{Ob_v}, \overrightarrow{Ag_v}, \overrightarrow{Sp_v}, \overrightarrow{Tp_v} \rangle$ consisting of one or more the five state vector element vectors $\overrightarrow{Bg}, \overrightarrow{Ob}, \overrightarrow{Ag}, \overrightarrow{Sp}, \overrightarrow{Tp}$. ■

As mentioned earlier, each of these elements is represented by an ontology hierarchy with the root as *StateVector* class. However, it should be kept in mind that although we argue that these five elements have been necessary to describe most world states it should be noted that the composition of the state vector space is not sufficient. In other words the definition of state vector space is not strict. The vector tuple $\langle \overrightarrow{Bg_v}, \overrightarrow{Ob_v}, \overrightarrow{Ag_v}, \overrightarrow{Sp_v}, \overrightarrow{Tp_v} \rangle$ simply specifies the complete format in which the classes have to be defined. Some of the

vectors in the tuple may have NULL value set according to the state vector space class definition that is required for a particular case. We now define a *state* in terms of an instance of the state vector space:

Definition 5.3: A *state* (denoted as $\vec{\varphi}$) is vector value tuple $\langle \overrightarrow{Bg_v}, \overrightarrow{Ob_v}, \overrightarrow{Ag_v}, \overrightarrow{Sp_v}, \overrightarrow{Tp_v} \rangle$ that comprises of the set of values taken by a state vector class ($\overrightarrow{SV}$) defined within a given domain. ■

The range of the values of a state $\overrightarrow{\varphi_\iota}$ depends on the particular domain in which it lies. This domain may be the WWW, pervasive systems or may be any generic distributed intelligent system. An activity can be associated to two types of states – *initial state* $\overrightarrow{\varphi_v^\iota}$ and *final state* $\overrightarrow{\varphi_v^f}$. The initial state is the system state that is changed due to the occurrence of the activity while the final state is the system state that is an effect of the activity. We now formally define an *activity* as follows:

Definition 5.4: An *activity* (denoted as ϑ) is an atomic process that when executed by an agent *a* till completion over a continuous time period Δ^T brings a change from an initial state $\overrightarrow{\varphi_v^\iota}$ to a final state $\overrightarrow{\varphi_v^f}$. ■

We can derive the following axiom from the above definition:

Axiom 5.1: The vector tuple comprising $\overrightarrow{\varphi_v^\iota}$ is equivalent to the vector tuple comprising$\overrightarrow{\varphi_v^f}$. ■

Thus, axiom 5.1 implies that there cannot be any vector element defining the initial state that does not define the final state when an activity occurs. In other words, we can say,

in corollary, that the activity defines the type of vector elements that comprises the state vector space class associated with the activity.

Agent cognitive ability needs to be defined formally so as to establish event cognition within a framework. In order to respond to a particular activity the agent needs perceptibility. This can be the capability to perceive (or sense) a particular activity via several modes such as IRs, surveillance cameras, temperature sensors, heat sensors, GPS-based location trackers, RFIDs, wearable mobile devices such as PDAs, smart pens, etc. Much of the raw data that is perceived is stream data. The raw data needs to be cleaned through data filtration techniques and then formatted into a specific feature vector that has a pre-defined semantics. The feature vector is then used to classify the perceived entity via Machine Learning techniques (such as image processing, speech recognition, etc). Other logical inference techniques (such as inductive reasoning, deductive reasoning, etc) are used to extract higher level knowledge regarding what is collected, aggregated and classified. The responsibility of data collection and processing can be given solely to the agents (assuming that the agents have such capability) or the processing job can be given to a higher level software module while the data collection is done by the agents. In this work we propose a state vector space based perceptibility definition where the pervasive device agents are required to extract the feature vectors in terms of *Bg*, *Ob*, *Ag*, *Sp*, and *Tp*. In fact, the vector tuple $\langle \overrightarrow{Bg_v}, \overrightarrow{Ob_v}, \overrightarrow{Ag_v}, \overrightarrow{Sp_v}, \overrightarrow{Tp_v} \rangle$ can be treated as a feature vector in itself. We define perceptibility as follows:

Definition 5.5: Perceptibility (denoted as $\mathbb{P}$) of an agent a is the $X_i \ni X_i \subseteq \overrightarrow{SV_j};\ \forall x \in X_i, \exists m \in M^a \ni x \xrightarrow{p} m$ set. ■

The implication of the above definition is that all the vector elements constituting the perceptibility vector tuple X_i should be mapped to at least one sensory mode of perception (the set of modes is denoted as M) of the agent a. We also can understand from the above definition that perceptibility of an agent is independent of the object of perception. One of major decisions that a pervasive system framework needs to take is whether to allow an agent perceive all the time or whether the agent should perceive only when some interesting activity is likely to happen. This is one of the most important issues regarding resource utilization as typically pervasive systems are not energy rich. Moreover, there is a growing concern in the research community to make the systems as *green* as possible. However, to facilitate such feature we need efficient mechanism for probabilistic reasoning. This reasoning can be based upon several evidences found within the system such as the behavioral habit of agents performing activities, the correlation between two or more activities, the causal effect of one or more activities over another activity, and so on.

While perceptibility is a necessary cognitive ability of an agent it is by no way sufficient. Agents need to understand the filtered data and take necessary actions based upon such understanding. For an example, a particular agent may perceive that a person has entered the room. It may also be able to classify the movement of the person as *entering room*. However, it may not understand what to do next as the classified information does not trigger any of its role axioms. Let us imagine that a role axiom is added to its KB that tells:

turn on the light if person enters the room. Now the classified information *entering room* actually triggers this role axiom. The cognitive ability to understand and respond to a particular activity is called *notability*. In order to formally define notability we first need to introduce an operator called the *sufficiently minimal subset operator* as follows:

Definition 5.6: A *sufficiently minimal subset operator* w.r.t. agent *a* (denoted as $\subseteq_M^a$) is defined as an unary operator over the initial state of an activity $\overrightarrow{\varphi_v^t}$ that produces the set $X = \left\{Y | (Y \subseteq \overrightarrow{\varphi_v^t}) \wedge \forall Y, \forall A \in U_a^R, (\nexists x \in Y, \exists\ \tau^x \xrightarrow{\not\models} A)\right\}$ where $\tau^x{}_{\tau} x$ is the transition from the initial state value to the final state value of the vector element *x*. ■

The above definition implies several things: (i) the operator is a functional operator and hence, produces an output, (ii) the output is a collection of sets (*X*) where each set *Y* is a subset of the initial state $\overrightarrow{\varphi_v^t}$ of the activity ϑ, (iii) each set *Y* is such that there is no vector element *x* comprising the set that does not trigger off any role axiom *A* in the set of role axioms U^A of the agent *a*. In other words, the set *X* contains all necessary and sufficient vector elements whose value transition invokes some role axiom in U^A.

An example in our case would be the event of *car renting* which is a *user activity*. In this case the user activity, as per *DQM*, is

$$D = \{\langle\{car\ profile, car\ confirmation\}, \{locatedIn(pickup, Kansas\ City) \wedge destination(Chicago)\rangle\}\}$$

$$I = \{\langle\{name, source\ location, destination\ location\}, \{null\}\rangle\}$$

where *D* is the user desire and *I* is the user input. According to *DQM*, if we consider only the desire part of the activity then the parameters *confirmation* and *car information* are the two background state variables that undergo change from an initial *empty* state to a final state

that takes on some values (such as *confirmed,* and *Honda Civic* respectively). Here the set *x* can be {*confirmation*}, {*car information*}, or {*confirmation, car information*}. Thus, it is to be noted that the set *x* is not a single possibility for interpretation with respect to a particular agent. There can be several versions of the set *x* (in the example three versions). Each version may constitute different state variable information such that all of them are minimal requirement for invoking an interpretation for a particular agent. In minimality of set *x* is required because otherwise the set will contain state variables that are unnecessary for any interpretation and hence, causes computational overhead for an agent to interpret an activity.

An agent can interpret an activity in several different ways depending on the set *x*. At the same time an agent can interpret an activity in the same way on different versions of the set *x*. For an example, by only observing at the set {*car information*} a *car lookup service* agent might interpret the consumer request activity to be that of *car lookup event*. While for some other *car rental agent* if the observation of the request activity is confined to only {*car information*} then it actually invokes no interpretation and hence, this agent will simply ignore the activity. We now formally define *notability* of an activity as follows:

Definition 5.7: *Notability* of an activity ϑ w.r.t. an agent *a* (denoted as $\aleph_v^a$) is defined as the set *Z* such that $Z \in \left\{Y \middle| Y = \subseteq_M^a . \overrightarrow{\varphi_v^i}\right\} \ni Z \neq \emptyset \wedge Z \xrightarrow{p} M$. ■

In the above definition we can conclude that: (i) notability set Z is a member set of the collection *X*, (ii) the set should not be empty, (iii) the notability set should be mapped to *M*, where *M* is the set of all perception modes within a pervasive system. From axiom 5.1 we can conclude that $\subseteq_M^a . \overrightarrow{\varphi_v^i} \equiv \subseteq_M^a . \overrightarrow{\varphi_v^f}$. Hence, we see that the definition of notability is

dependent on the state (initial or final) of the activity. However, not all the vector elements composing the activity state are necessary and sufficient for triggering a role axiom in an agent. Only those elements are selected that are necessary and sufficient and also perceptible by the system. We can also deduce from the definition that notability of an activity by an agent may not be the same as the agent's perceptibility. For an example, the agent that turns the light on when a person enters the room may not have the perceptibility of the activity *entering the room*. However, the activity can still be notable by the agent if the activity is perceptible by some other agent/s and the perception is somehow accessible to the agent.

The notion of notability defines the relation between *activity* and *event* as has been explicated in the following definition.

Definition 5.8: An *event* w.r.t. an agent *a* (denoted as ε^a) is any activity ϑ which has a corresponding notability $\aleph_v^a$ with respect to an agent *a*.■

A major advantage of defining *events* in this manner is that, unlike conventional ED-SOA systems, *events* do not have to be pre-defined and then published or subscribed into the system. Instead the system lets its constituent services to capture different interpretations of the same activity and register them into a common knowledge base. Dynamic registration of different events is based on the different notability sets ($\aleph_v^a$) that are detected by the different devices (i.e. *a*) of the system.

The *event notability theory*, thus, sets the platform for a dynamic pro-active approach of understanding and responding to events by devices hosting services (or agents). Service discovery is therefore also proactive in the sense that it is the services that discover (from an event interpretation point of view) the query rather than the query discovering the services.

However, in this section we did not discuss the inter-dependency of these agents when they have to work in cooperation to both interpret as well as respond to a particular system activity (user activity as well as service). This is an extremely important problem to study because in many cases agents are causally related to each other from a functional standpoint. If one agent executes a service then it causes a set of interpretations for some other agents that may trigger them to execute in turn. The interpretation of an activity by an agent, in this way, may be dependent on the interpretations of the same activity by other agents. In the next section a unified middleware architecture called *ALNet* is proposed that helps individual service agents to interpret user as well as other service events and lets them respond according to their corresponding interpretations.

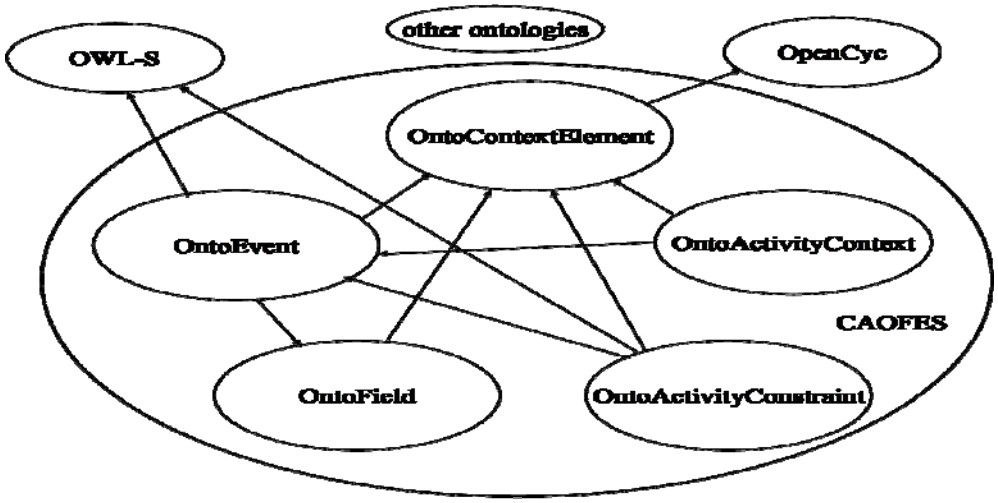


Figure 33: CAOFES - Top Level Scheme

5.3.2 Context-Aware Ontology Framework for Events and Services (CAOFES)

In the previous section we discussed the essential formalism required for understanding a service event and a user event with respect to the agents within a web

domain. However, these formalisms require a semantic foundation in order to conduct efficient semantic query processing. That is, the concepts so defined in section 6.3 must have semantic definitional equivalency. In order to provide such a platform we developed a framework called *Context-Aware Ontology Framework for Events and Services* (*CAOFES*). CAOFES is a collection of five core ontologies: (i) *Context Element Ontology* (ii) *Activity Context ontology*, (iii) *Activity Constraint ontology*, (iv) *Event ontology*, and (v) *Field ontology* (figure 33). Apart from these defined ontologies we use the primal ontology OWL-S for describing the service functional vector $\vec{P}$. Other upper level ontologies such as OpenCyc [201] can be imported as the framework grows and equivalent concepts are needed to be incorporated in the core ontologies. We now formally describe each of the four ontologies in terms of Description Logics (DL) notations.

5.3.2.1 Context Element Ontology

The Context Element Ontology (named as *OntoContextElement*) is a DL terminology that houses the semantic definitions of the space vector elements $\langle \overrightarrow{Bg_v}, \overrightarrow{Ob_v}, \overrightarrow{Ag_v}, \overrightarrow{Sp_v}, \overrightarrow{Tp_v} \rangle$ that has been defined in section 5.3.1. The root concept *Context Element* (*CE*) is defined as a primitive concept. It can be further classified into the five corresponding vector element concepts *Bg*, *Ob*, *Ag*, *Sp*, *Tp*. Each of these concepts can be further classified. It is to be noted that the concepts are generally borrowed or derived from upper level ontologies such as OpenCyc, temporal ontologies, spatial ontologies and the like. OntoContextElement is the building block of the other core ontologies defined within CAOFES as all other core ontologies can be related to each other through this ontology.

5.3.2.2 Activity Context Ontology

The Activity Context Ontology (named as *OntoActivityContext*) is a DL terminology that houses the semantic definitions of concepts related to the vector elements $\langle\overrightarrow{Bg_v}, \overrightarrow{Ob_v}, \overrightarrow{Ag_v}, \overrightarrow{Sp_v}, \overrightarrow{Tp_v}\rangle$defined in vector space $\overrightarrow{SV}$. The corresponding semantic interpretation of $\overrightarrow{SV}$ called context vector space (denoted as *CV*) is:

$$CV \equiv (\exists hasElement\,.\,(Bg \sqcup Ob \sqcup Ag \sqcup Sp \sqcup Tp))$$

where *Bg*, *Ob*, *Ag*, *Sp*, *Tp* corresponds to the vector elements.

The existential quantifier in the above definition leaves it open for the ontology designer to introduce other filler concepts such as QoS parameter concepts like network latency, throughout, bit error rate etc. for more specific application domains. CV is the root concept of OntoActivityContext. The instances of CV are the *states*. We define a necessary condition for *A* as: $A \equiv \forall executes\,.\,E$ where *E* is the event concept defined in the Event Ontology.

The context vector concept CV can be further classified into the concept *notability vector* (denoted as NV), and the concept *event context vector* (denoted as ECV). We defined each of them as:

$$NV \equiv (\sqsubseteq.CV) \sqcap (\forall isNotedBy\,.\,E) \sqcap (\forall notabilityOf\,.\,E).$$

$$ECV \equiv (\sqsubseteq.CV) \sqcap \big(\forall isInterpretationOf\,.\,(CCN \sqcup SCN\,)\big) \sqcap (\forall contextOf.E)$$

where *CCN* is the causal constraint and *SCN* is the state constraint defined in the Activity Constraint ontology (see section 5.3.2.3). *NV* is the semantic equivalent of $\aleph_v^a$. *ECV* is the semantic equivalent of $\overrightarrow{SV}$.

OntoActivityContext can be further classified as per the domain requirement keeping the given definitions consistent.

5.3.2.3 Activity Constraint Ontology

The Activity Constraint Ontology (named as *OntoActivityConstraint*) is a DL terminology that comprises of the semantic definitions of the different kind of constraints that govern the behavior of agents executing events. We define the root concept of the ontology, constraint (denoted as *CN*), as follows:

$$CN \equiv \exists \geq 1\ hasClause\ .(Bg \sqcup Ob \sqcup Ag \sqcup Sp \sqcup Tp)$$

The above definition reflects the fact that the domain of interpretation of the constraints for any event is the same as the context vector elements. In other words, constraints can be specified in terms of concepts derived from one or more of these five vector elements. We assume that constraints are always formalized in the CNF (Conjunctive Normal Form). The CN concept can be further classified into three kinds: (i) causal constraint (denoted as CCN), (ii) state constraint (denoted as SCN), and (iii) stimulus constraint (denoted as ZCN).

The CCN concept specifies the causal condition required for an event to be an effect of other event(s). We define CCN as follows:

$$CCN \equiv (\sqsubseteq . CN) \sqcap (\forall isCausalConstraintOf\ .E).$$

The SCN concept specifies the initial state condition required to be satisfied for an event to occur. We define SCN as follows:

$$SCN \equiv (\sqsubseteq . CN) \sqcap (\forall isStateConstraintOf\ .E).$$

The ZCN concept specifies the post-conditional stimuli given by a causal event in order to trigger other event(s) as its effect. We define ZCN as follows:

$$ZCN \equiv (\sqsubseteq . CN) \sqcap (\forall isStimulusContraintOf\ .E) \sqcap (\forall \geq 1\ . satisfies. CCN).$$

From the above definition we can see that ZCN is a model concept of the concept CCN. Hence, any constraint instance that satisfies the definition connects two events with a causal relation – the one that is an interpretation of the ZCN definition as the cause and the one that is an interpretation of CCN definition as the effect.

5.3.2.4 Event Ontology

The Event Ontology (named as *OntoEvent*) is a DL terminology that comprises of the different kind of events including user events and services that may be identified within a particular system (or field). The root concept *event* (denoted as *E*) is defined as follows:

$$E \equiv (\forall hasInitialState\ . CV) \sqcap (\forall\ hasFinalState. CV) \sqcap (executedIn. F) \sqcap (hasConstraint\ . CN)$$

where *F* is the concept *field* defined in Field Ontology (see section 5.3.2.5). E is semantic equivalent of ε^a.

The concept E can be further classified into two concepts: (i) user event (denoted as UE) and (ii) service event (denoted as SE). We define each of them as follows:

$$UE \equiv (\sqsubseteq . E) \sqcap (\forall executedIn\ . EF) \sqcap (\forall\ triggers. SE)$$

where *EF* is the *environment field* defined in Field ontology. UE is semantic equivalent of ε_u^a.

$$SE \equiv (\sqsubseteq . E) \sqcap (\forall executedIn\ . SF) \sqcap (\forall\ hasFunctionality. P)$$

where *SF* is the *system field* defined in Field ontology. S is the semantic equivalent of ε_u^a. *P* is the semantic equivalent of the service functional vector $\vec{P}$. We define the concept P as follows:

$$P \equiv \forall\, hasElements.\,(In \sqcup Ou \sqcup Pr \sqcup Re)$$

where *In*, *Ou*, *Pr*, *Re* are the service profile concepts defined in OWL-S.

OntoEvent connects itself to the agents that interpret it. Hence, it helps the reasoner to understand the relation between events (both user query and services) for deducing possible dependency. The moment an event instance is recognized as *UE* the CAOFES automatically defines it by connecting the event to the *notability vector concept* (*NV*) defined in OntoActivityContext. This definition connects the event to all the context vector elements that are needed for defining the notability and also to the agent(s) (and hence, the service) that interprets it. Hence, a causal definition is established between an *UE* and *SE* if and only iff the antecedent of the following rule is valid:

Rule of User Event – Service Event Dependency:

$$HasNotability\big(UE(u), NV(n)\big) \wedge isNotedBy\big(NV(n), SE(s)\big) \wedge satisfies\left(ZCN\big(UE(u)\big), CCN\big(SE(s)\big)\right) \leftrightarrow triggers(UE(u), SE(s))$$

The predicates defined in the above rule correspond to the relations defined in the above mentioned ontologies.

5.3.2.5 Field Ontology

The Field Ontology (named as *OntoField*) is a DL terminology that defines and classifies the world in which events occur. The root concept *field* (denoted as *F*) is defined as follows:

$F \equiv \forall\, hasExecuters.Ag$ where Ag is Agent defined in section 6.4.2.

The concept *F* can be further classified into two sub concepts: (i) *system field* (denoted as *SF*) and (ii) *environment field* (denoted as *EF*). We define the each of them as follows:

$SF \equiv (\sqsubseteq .F) \sqcap (\forall hasExecuters.D)$where *D* is the service hosting device agent s.t. $D \equiv \sqsubseteq .Ag$.

$EF \equiv (\sqsubseteq .F) \sqcap (\forall hasExecuters.(U \sqcup D))$ where U is the user agent s.t. $U \equiv \sqsubseteq .Ag$.

Ontofield connects OntoEvent with the agents executing specific events defined. It further helps to reason about the type of event (whether a user event or a service event) by verifying the type of executer.

5.4 Activity Logic Network (ALNet): Conceptual Foundation

As mentioned before an event-driven SOA system can be modeled as a causal network of events (services and user events) called *Activity Logic Network* (*ALNet*). *ALNet* can also be viewed as a logical workflow of events [74 - 76, 104 - 105, 202]. The nodes of *ALNet* are *event vectors* (both services and user-events). Although, in general, *nodes* in ALNet may signify both service and user event vectors we restrict the definition of nodes only to *service vectors* that within the system (figure 34). We define *service vectors* as follows:

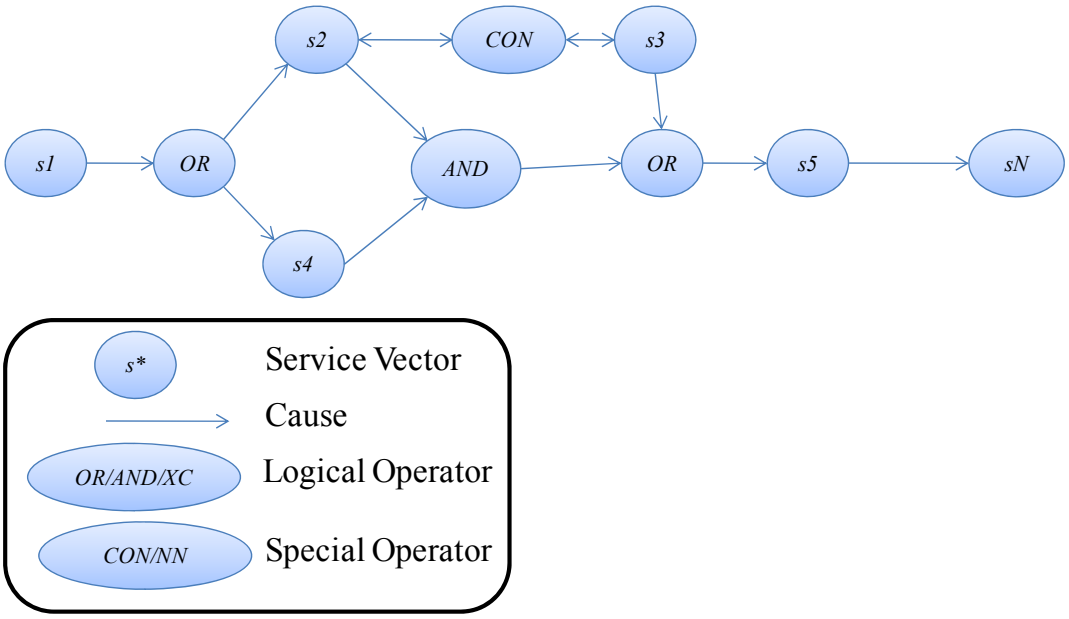


Figure 34: An *ALNet* instance

Definition 5.9: *Service vector* is a 4-dimensional vector tuple $\langle \overrightarrow{In}, \overrightarrow{Ou}, \overrightarrow{Pr}, \overrightarrow{Re} \rangle$ where the vector elements denotes: (i) input (denoted as $\overrightarrow{In}$) that specifies the input parameters of a service, (ii) output (denoted as $\overrightarrow{Ou}$ that specifies the output parameters of a service, (iii) pre-condition (denoted as $\overrightarrow{Pr}$) that specifies system state that are needed to be satisfied for a service to be invoked, and (iv) result (denoted as $\overrightarrow{Re}$) that specifies the system state that is generated as a result of the service execution.■

This sort of treatment allows us to perform a variety of operations over *ALNet* including service discovery and composition. It is to be noted that a *service event* is the scalar equivalent of the *service vector*. As user-events are assumed to be handled solely by services hence, we assume that the event-handling process occurs only within the system. An *edge* in the ALNet (figure 34) signifies the causal relation between two services where one of the services is the functional *cause* and the other service is the functional *effect*. The

effect service responds to the *stimuli* given by the *causal service*. We term the effect service as *responsive service*. A stimuli generated by a causal service is the result vector element $\overrightarrow{Re}$ of that service. For a causality to be established the stimulus $\overrightarrow{Re}$ should be perceived by the responsive service and it should satisfy the causality constraint (i.e. the vector element $\overrightarrow{Pr}$) of the responsive service. A stimulus from a causal service can be perceived by a responsive service only if there exists a communication channel between the two. A detailed discussion on service to service communication is given in the next section.

If causality is established between two services then we say that the two services have *dependency*. *Dependency* (denoted as ←) can be two types: (i) *strong dependency* and (ii) *weak dependency*. We define each as follows:

Definition 5.10: A *strong dependency* is said to exist between a causal service vector s_1 and a responsive service vector s_2 (denoted as $s_1 \overset{s}{\leftarrow} s_2$) iff $\exists x, y \ni (x \in s_2.input, y \in s_1.output) \wedge (\overset{g}{\supset}(x, y) \in \{exact, subsume\})$ where $\overset{g}{\supset}$ is the *g-subsumption* operator (chapter 3). ■

Definition 5.11: A *weak dependency* is said to exist between a causal service vector s_1 and a responsive service vector s_2 (denoted as $s_1 \overset{w}{\leftarrow} s_2$) iff $\exists x, y \ni (x \in s_2.input, y \in s_1.output) \wedge (\overset{g}{\supset}(x, y) \in \{plugin, sibling\})$. ■

We represent *dependency* with two mutually inverse functions: (i) the function *CA* where *CA* denotes *causal* and (ii) the function *RE* where *RE* denotes *responsive*. The function takes three arguments: (i) a service vector (causal or responsive), (ii) the precondition formula θ, and (iii) the system $\sum$. We use a predicate *equals* (denoted by *eq*) that takes three arguments: (i) a dependency function (CA or RE), (ii), a service vector (causal or responsive) and (iii) time variable *t*.

The following axiom relates the CA function and the RE function:

Axiom of dependency equivalence: $eq\left(CA\left(\varepsilon_j, \Theta_p, \Sigma_q\right), \varepsilon_i, t\right) \leftrightarrow\ eq\left(RE\left(\varepsilon_i, \Theta_p, \Sigma_q\right), \varepsilon_j, t\right)$■

It should be noted that a particular responsive service may have more than one causal service and a particular causal service may have more than one responsive service. This is the case where the dependency is partial. We define *partial dependency* as follows:

Definition 5.12: A *partial dependency* is said to exist between a causal service vector s_1 and a responsive service vector s_2 iff $\exists x \in s_2.input\ \ni\ \forall y \in s_1.output;\ (\overset{g}{\supset}(x,y) \in \{fail\})$. ■

In some cases the multiplicity in causality may mean a conjunction (i.e. a logical AND) where a particular responsive service can trigger off only if *all* its causal services are executed (figure 34). In other cases multiplicity in causality may mean a disjunction (i.e. a logical OR; figure 34). In this case a particular responsive service can trigger off if *any* of its causal services are executed. The third type of cases is when a multiplicity in causality means an exclusive disjunction (i.e. an n-ary logical operator called XC). XC is an n-ary operator that is semantically equivalent to *only one*. The above mentioned arguments go the

same for multiplicity in effects. For multiplicity in causality we use the prefix *pre* before the operators and for multiplicity in effect we use the prefix *post*. We denote logical operators in general as *L* and we use the superscripts *pr* and *po* to imply whether that is a pre or a post. Apart from the logical operators there are two special operators called *Conflict* (denoted as *CN*) and *Not Necessary* (denoted as *NN*). The *CN* operator explicitly states the conflicting effect of two services. If a service s_i is connected to another service s_j via a *CN* within *ALNet* then it means that the occurrence of the former should eliminate any possible occurrence of the latter until the former finishes execution. Thus, *CN* is basically an restriction to the selection of the service S_j during a particular event-handling process. This does not necessarily mean that S_j cannot be selected for some other event-handling process that may happen after the current event-handling process terminates. On the other hand the NN operator helps us to model event-handling processes where feedbacks are existent. By feedback we mean dual communication between two services S_i and S_j where S_i gets occasional feedback information (or stimulus) from S_j but happens to be the causal service to S_j. In this case S_j is not a necessary causality for S_i (otherwise there would have been a causal deadlock) but occasionally causes S_i to respond to its stimulus. We denote the special operators in general as *Q*. We summarize the descriptions in table 6.

Edges in an ALNet can be of two types: (i) *constrained edge* (denoted as E^{θ}) and (ii) *non-constrained edge* (denoted as $E^{\emptyset}$). A *constrained edge* is one which represents a dependency between a causal service and a responsive service and hence, a causality constraint/stimulus is implied over it. On the other hand, a *non-constrained edge* is one that is only a connective between two logical nodes or between a logical node and a service node

in ALNet and does not imply any direct dependency between the nodes that it connects together. Hence, no causality constraint/stimulus is implied over it. We now present the formal definition of ALNet as follows:

Table 6: ALNet Operators – Semantics and Symbols

ALNet pre Functions	**ALNet Expressions**	**Symbol**	**Explanation**
Pre AND	$pre\_AND(A_i, C_{ij}, E_k) = a_j$	$\wedge^{pre}$	a_j is triggered by all the members of the set A_i provided that the set of constraints C_{ij} is satisfied under the environment E_k
Pre OR	$pre\_OR(A_i, C_{ij}, E_k) = a_j$	$\vee^{pre}$	a_j is triggered by at least one of the members of the set A_i provided that the set of constraints C_{ij} is satisfied under the environment E_k
Pre XClusive	$pre\_XC(A_i, C_{ij}, E_k) = a_j$	$\circ^{pre}$	a_j is triggered by only one of the members of the set A_i provided that the set of constraints C_{ij} is satisfied under the environment E_k
ALNet post Functions	**ALNet Expressions**	**Symbol**	**Explanation**
Post AND	$post\_AND(a_i, C^I_{ij}, E_k) = A_j$	$\wedge^{po}$	a_j *can possibly* trigger all the members of the set A_j provided that all the constraints of the set of internal logic constraints C^I_{ij} are satisfied under the environment E_k
Post OR	$post\_OR(a_i, C^I_{ij}, E_k) = A_j$	$\vee^{po}$	a_j *can possibly* trigger at least one member of set A_j provided that at least one corresponding constraint of the set of internal logic constraints C^I_{ij} is satisfied under the environment E_k
Post XClusive	$post\_XC(a_i, C^I_{ij}, E_k) = A_j$	$\circ^{po}$	a_j *can possibly* trigger only one member of set A_j provided that only one corresponding constraint of the set of internal logic constraints C^I_{ij} is satisfied under the environment E_k
ALC\Net Special Functions	**ALNet Expressions**	**Symbol**	**Explanation**
Conflict	$CON(a_i, C_{ij}, E_k) = a_j$	CN	a_j cannot be triggered if a_i has already occurred under the environment E_k
NOT Necessary	$NN(a_i, C_{ij}, E_k) = a_j$	NN	a_i is not a necessary causality but if executed will trigger a_j provided that the set of constraints C_{ij} is satisfied under the environment E_k

Definition 5.13: An *ALNet* can be defined by the set A^L s.t. $A^L = \{\langle V, E\rangle |\, V = (\vec{S} \cup L \cup Q) \ni \left(L^{pr} \times V \to E^{\emptyset}\right) \wedge (V \times L^{po} \to E^{\emptyset})\}$ where $L = L^{pr} \cup L^{po}; V \times V \to E; \vec{S} = \langle \overrightarrow{C_S}, \vec{P}\rangle; Q = \{CN, NN\}; E = E^{\emptyset} \cup E^{\Theta}$.

It can be noted from the above definition that the edges E between any node and the L nodes in ALNet is restricted in two cases. In the first case there cannot be any constrained edge from an L^{pr} node to any other node. In the second case there cannot be any constrained edge from any node to an L^{po} node in ALNet. As the pre-operator L^{pr} is a connective between multiple causal service nodes to a single responsive service node hence, the causal constraint of the responsive node should not be placed over the edge connecting the pre-operator to the responsive node. Instead, the causality constraint is split into multiple causality constraints and divided over each of the multiple edges coming from the causal service nodes to the pre-operator. The nature of the split (i.e. a conjunctive split or a disjunctive split) decides the nature of the pre-operator. The same argument goes with the post-operator L^{po} that connects a single causal service node to multiple responsive service nodes. Hence, the stimulus of the causal node should not be placed over the edge connecting the causal node and the post-operator. Instead it is split and divided over the edges connecting the post-operator to the multiple responsive nodes. The nature of the post-operator is decided by the nature of the split. There are 16 different services that can be identified. These services are causally related to each other. We see that there are 9 logical operators out of which 7 are post-AND and 2 are pre-OR. There are 6 post-AND operators that connect the user events with 10 source services (i.e. the effects).

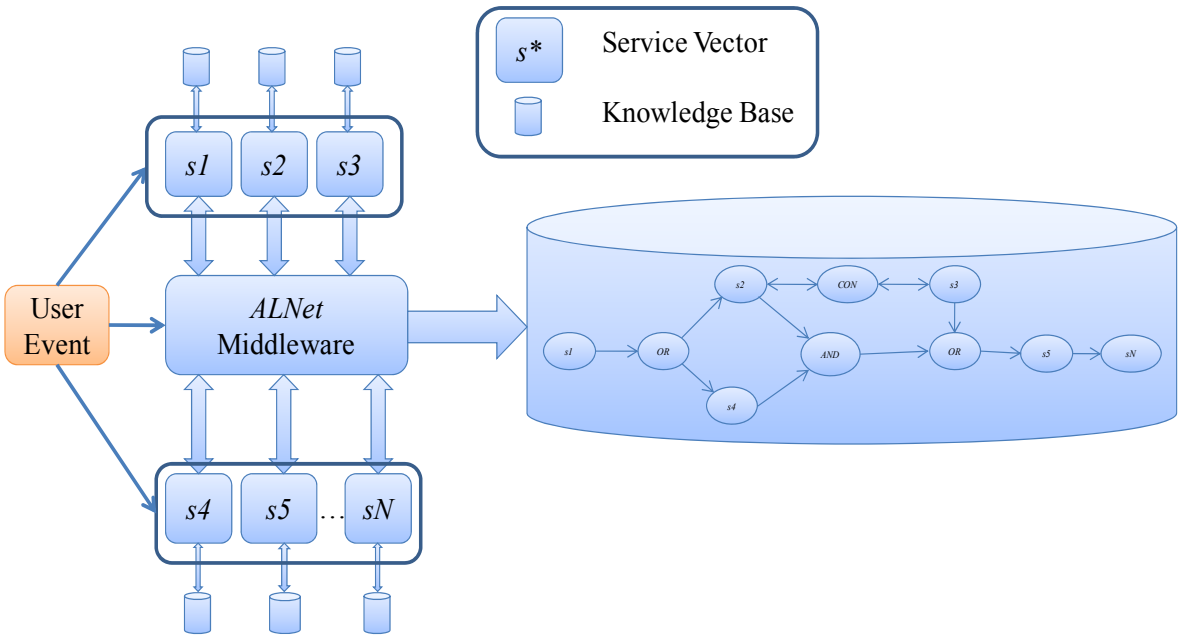


Figure 35: *ALNet* architecture

5.5 *ALNet* Architecture

In this section the architectural backbone of ALNet is introduced. The architecture consists of the centralized *ALNet* middleware and the set of service descriptions that are registered with the middleware (figure 35). It is to be understood that the set of services is dynamic in the sense that there can be addition and deletion of services from the set over time. Service descriptions are transformed into corresponding service vectors and the mutual functional dependency between these service vectors is dynamically generated. Thus, the underlying *ALNet* instance grows and shrinks over time as well. The *ALNet* middleware maintains a global view of the overall *ALNet* in its knowledge base. At the same time each service vector that is registered with the middleware maintains in its own local knowledge

base a local *ALNet* instance that contains the service vector's own functional dependency with other service vectors.

Apart from the *ALNet* instances each service vector and the middleware maintain their corresponding version of *CAOFES* instances (introduced in section 5.3). A *CAOFES* instance acts as a semantic belief system for a service agent (and the *ALNet* middleware) that enables the service agent to semantically interpret events (services and user events). The interpretation of a service stimulus as an event by a service agent results in the possibility of execution of the service by the agent causing, in turn, stimuli for other possible agents within the system. Interpretation of service stimuli can be re-active where the *ALNet* middleware (figure 35) invokes a set of service agents by interpreting the stimuli on behalf of them or pro-active where agents have defined sensory modes. In the case of reactive response a set of service agents are notified by the middleware based on a match of the user request *desire* with the *output* of the service vector corresponding to each of those service agents. In other words the middleware interprets the user event on behalf of the service agents. On the other hand in pro-active response service agents perceive and interpret the user agent themselves. Based on a particular interpretation a service agent may generate stimuli that is either pro-actively captured by other service agents or reactively responded to via the middleware. Sometimes both the service agents as well as the middleware can participate collaboratively into the interpretation of service stimuli. This happens when a user agent may not have all the necessary sensory modes or the complete belief system for fully interpreting the event.

The *ALNet* architecture allows asynchronous execution of services that is based on a completely independent order of event interpretation and responses. The middleware keeps record of all events (both user events and services) that occur within the system. Thus, even though services may be stateless they need not be synced together for interpreting each other's stimuli. Although a service agent may be busy over a time period yet its corresponding service vector is able to interpret an event and keep the interpretation with the middleware. The middleware then notifies the service agent when it is free so that the service agent can respond according to the interpretation. In other words, the interpretation of an event is decoupled from the response to that event within the *ALNet* architecture.

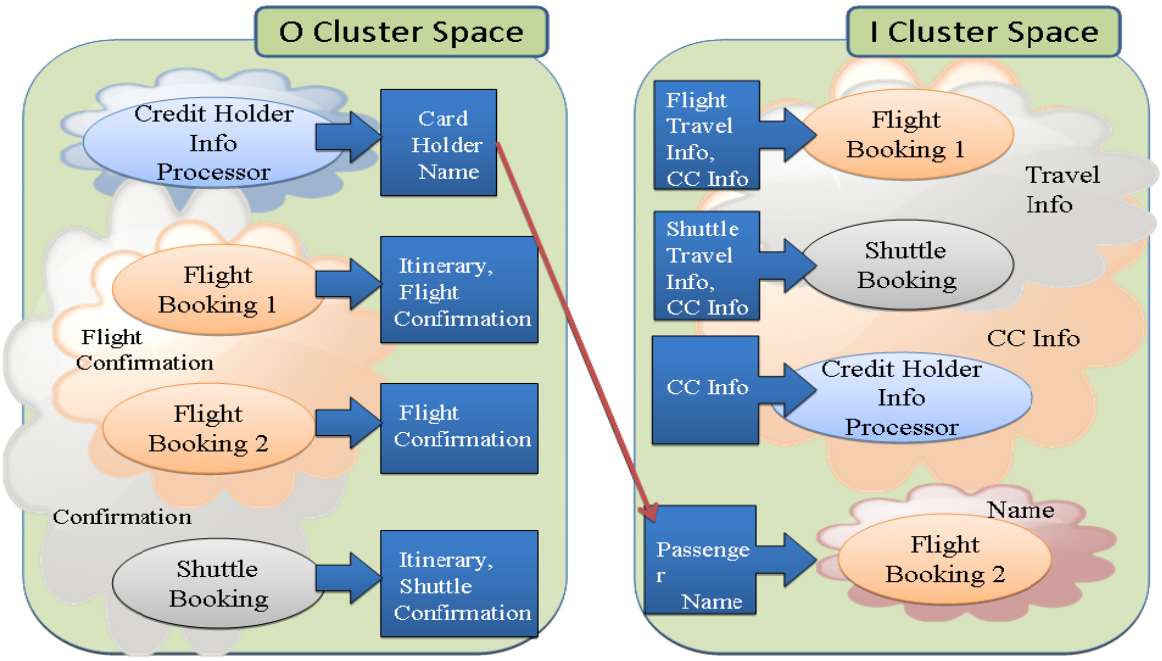


Figure 36: Abstract Edge between Clusters in O-cluster space and I-cluster space

5.5.1 ALNet Abstraction

As can be understood by the definition of *ALNet*, with the addition of more and more services (especially functionally similar services) the number of service dependency defined will grow. Hence, the ALNet becomes more complex as it grows with time. Thus, the problem of service discovery becomes much more complicated and computationally expensive due to the potential explosion of the search space. In order to resolve this issue efficiently we propose an abstraction technique termed *ALNet Abstraction* that transforms a potentially complex *ALNet* into a simpler *abstract ALNet*.

ALNet abstraction leverages upon the *STC clustering algorithm* (proposed in chapter 4) that groups functionally similar service vectors into taxonomies (i.e. clusters). We term such a cluster as an *abstract node* (denoted S^A) in the context of *ALNet*. Hence, definitionally *abstract node* and *g-taxonomy* are equivalent. The root service vector of a cluster describes the *abstract node* since semantically it subsumes all other service vectors in that cluster. As *STC* generates two mutually independent cluster spaces (*O-cluster space* and *I-cluster space*) we therefore have two corresponding abstract nodes - *O-abstract node* (where corresponding $g = O$) and *I-abstract node* (where corresponding $g = I$). After the abstract nodes are generated the *O-cluster space* is mapped onto the *I-cluster space* such that for a particular *O-cluster* root service vector a *dependency* can be established with at least another *I-cluster* service vector (figure 36). This is done by selecting each *O-cluster* in the *O-cluster space* and then searching for the *I-cluster* in the *I-cluster space* such that there exists at least one service vector (say *s*) in the *I-cluster* that is in *LSC* (i.e. *Least Specific Children* set in the context of *g-subsumption*) with the root service vector of the *O-cluster*.

After this operation is done all the ancestor and descendant service vectors of *s* are extracted to form an *abstract responsive node*. The corresponding *O-cluster* is the *abstract causal node*. Note that during this operation there can be several *abstract responsive nodes* corresponding to one *abstract causal node* since the *LSC* set may have more than one service vector. All such responsive nodes are causally dependent on the causal abstract node via a logical AND node. The dependency relation is represented as an *abstract edge*. We define as *abstract edge* as follows:

Definition 5.14: An *abstract edge* (denoted as E^A) is defined as an edge E^A_{ij} such that $\forall i,j; E^A_{ij} = V_i \times V_j; i \neq j; V = \{S^A, L, Q\} \ni [\exists k \in \{i,j\}; V_k = S^A]$. ■

According to the above definition an abstract edge can hold between three types of nodes: (i) abstract service node, (ii) AND nodes, and (iii) special nodes. Hence, there can be four kinds of abstract edges: (i) $S^A \times S^A$ (ii) $S^A \times (AND/Q)$ (iii) $(AND/Q) \times S^A$ and, (iv)$(AND/Q) \times (AND/Q)$. It is to be noted that an *abstract edge* is without any constraint imposition by definition. It is to be noted that the abstracted *ALNet* is not a global structure in the sense that the algorithm does not guarantee an abstraction over the entire *ALNet* instance to form a single *abstract ALNet* instance. Instead, the abstraction is at a local level where each service vector after undergoing an *STC* operation gets a unique identity of the root service vector/s of the clusters in which it belongs (note that according to *STC* there can be multiple cluster membership). If the root service vector has an *I-abstract* responsive node then all the service vectors within that cluster has a dependency with the abstract responsive node. The abstract responsive node is represented by the corresponding root service vectors and hence, gets the unique ID of each of these root service vectors. The local abstract

dependency is preserved within the individual knowledge bases of the service vectors. The *ALNet abstraction* algorithm is given as in figure 37.

```
ALGORITHM: ALNet Abstraction
INPUT: I-cluster space, O-cluster space
OUTPUT: Abstract ALNet
START
For each O-cluster in O-cluster space
        LSC_s_root = findLSC(s_root, I-cluster space)
        If LSC_s_root != NULL
                For each s_i in LSC_s_root
                        s_root._effect = [s_root._effect] ∪ [Ancestor(s_i) + Descendent(s_i)]
                End Loop
        End If
        root_effect = findRoot(s_root.effect)
        For each s_j in root_effect
                s_root.abstract_effect = s_root.abstract_effect L s_root // where L is an AND/OR
                s_j.abstract_cause = s_j.abstract_cause L s_root
        End Loop
End Loop
END
```

Figure 37: ALNet Abstraction Algorithm

ALNet abstraction algorithm has some very interesting and important properties in the context of service discovery and composition as follows:

Theorem 5.1: If a root service vector s_{root} of an abstract causal node S_{causal} has dependency with another abstract responsive node $S_{responsive}$ then each service vectors within S_{causal} has dependency with all other service vectors within $S_{responsive}$.

Proof: If s_{root} has an LSC and/or MSP in $S_{responsive}$ then it has *g-relation* with all the member service vectors of both LSC and MSP. Hence, it has g-relation with all the ancestors of each *MSP* member and all the descendants of each member of *LSC*. Since s_{root} *g-*

subsumes every other service vectors in S_{causal} therefore s_{root} has a *g-relation* with all the service vectors in $S_{responsive}$ As g-relation is an order relation hence, every service in S_{causal} has *g-relation* with all the members of $S_{responsive}$. ■

Corollary 5.1: If a root service vector s_{root} of an abstract node S_i does not have dependency with another abstract responsive node S_j then each service vectors within S_i does not have dependency with any other service vectors within S_j

Proof: If s_{root} does not have dependency with another abstract node S_j then it does not have any MSP or LSC set with respect to S_j. Thus, all other service vectors in S_i has no *g-relation* within any of the members of S_j. ■

The above theorems imply the soundness and completeness of the *ALNet Abstraction* algorithm.

5.5.2 ALNetSniffer: 2-Phase Service Discovery

As has been outlined in chapter 3, *ALNetSniffer* consists of two important phases: (i) end service discovery (this is equivalent to the *desire matching phase* of the *DQM* model), and (ii) source service discovery (this is equivalent to the *input matching phase* of the *DQM* model). Each matching phase generates a set of *weak solution set* (denoted as *WSS*) and a *strong solution set* (denoted as *SSS*) after the operation. We define each set as follows:

Definition 5.15 (*Weak Solution Set*): A *Weak Solution Set* w.r.t to a given query Q is the set of all services (denoted as *WSS*) such that $\forall s_i \in WSS; s_i \vDash Q \; where \; \vDash \begin{cases} \stackrel{O}{\sqsupseteq} if \; Q \; is \; Type \; 1 \\ \stackrel{I}{\sqsubseteq} if \; Q \; is \; Type \; 2 \end{cases}$ ■

Definition 5.16 (*Strong Solution Set*): A *Strong Solution Set* w.r.t to a given query Q is the set of all services (denoted as *SSS*) such that $\forall s_i \in SSS; s_i \vDash Q \; where \vDash \begin{cases} \overset{O}{\sqsubseteq} if\ Q\ is\ Type\ 1 \\ \overset{I}{\sqsupseteq} if\ Q\ is\ Type\ 2 \end{cases}$ ■

We now describe each phase of ALNetSniffer in detail as follows:

Phase 1: The user Q-T1 query event is mapped over the *abstract causal nodes* in the *abstract ALNet* so as to insert the query into the space in the same way as how services are clustered by the *STC* algorithm. More specifically an MSP of the query is first searched and then an LSC of the query is extracted out of the children set of the MSP. After that the MSP and its ancestors becomes the *weak solution set* (denoted *WSS-P1*) of the Q-T1 query while the LSC and its descendants become the *strong solution set* (denoted *SSS-P1*). The phase 1 combined solution set is called *possible end services*. The corresponding algorithm is given in figure 38.

```
ALGORITHM: ALNetSniffer-Phase1
INPUT: Q-T1, O-cluster space
OUTPUT: {WSS-P1, SSS-P1}
START
WSS-P1 = SSS-P1 = NULL // initially the solution sets are empty
sample = Q-T1
MSP = findMSP(sample, O-cluster space);
for 1 to MSP.size
{
        PLSC = PLSC ∪ findLSC(memberOf(MSP)); // PLSC: Potential LSC
}
findLSC(sample, PLSC, O-cluster space);
WSS-P1 = MSP ∪ Ancestor(MSP)
SSS-P1 = LSC ∪ Descendant(LSC)
return (WSS-P1, SSS-P1);
END
```

Figure 38: ALNetSniffer-Phase 1 algorithm

Phase 2: After the phase one is done the phase 2 is initiated only if either one of the WSS and the SSS so found in phase 1 is not empty. This is because if the phase 1 generates empty solution then the desire cannot be satisfied by the system at the time of query mapping. Hence, if at least one of these solution sets are non-empty then phase 2 starts off. In phase 2 the Q-T2 query event is mapped over the all the *abstract I-cluster nodes* in the *abstract ALNet* in the same way as in phase 1. However, after the MSP and the LSC have been discovered for Q-T2 the *WSS-P2* consists of the LSC and all its descendants (in contrast to phase 1) while the *SSS-P2* consists of the MSP and all its ancestors. The corresponding algorithm is given in figure 39.

```
ALGORITHM: ALNetSniffer-Phase2
INPUT: Q-T2, I-cluster space
OUTPUT: {WSS-P2, SSS-P2}
START
WSS-P2 = SSS-P2 = NULL // initially the solution sets are empty
sample = Q-T2
MSP = findMSP(sample, I-cluster space);
for 1 to MSP.size
{
        PLSC = PLSC ∪ findLSC(memberOf(MSP)); // PLSC: Potential LSC
}
findLSC(sample, PLSC, I-cluster space);
SSS-P2 = MSP ∪ Ancestor(MSP)
WSS-P2 = LSC ∪ Descendant(LSC)
return (WSS-P2, SSS-P2);
END
```

Figure 39: *ALNetSniffer-Phase 2* algorithm

Note that the solution sets in phase 2 may not be corresponding to a particular *I-cluster abstract node*. In fact the combined solution set is a subset of such a node. Also it

should be noted that the combined solution set may not be equivalent to an *abstract responsive node* since the responsive set must have a causal dependency with another *abstract causal node* while the combined solution set does not guarantee that. In certain cases the combined solution set of phase 1 and the combined solution set of phase 2 may have intersection in the sense that some of the service vectors are capable to generate the desired output as well as consume the given input. In a more specific case where the intersection is equivalent to the union query mapping becomes a one-to-one process. Although this has been assumed by most researches in the area of service discovery and composition we clearly see that the assumption cannot be generalized. For service discovery to be accurate a 2-phase approach such as *ALNetSniffer* is able to capture cases where there can be service vectors in the phase 1 combined solution set that are absent from the intersection (if that exists at all) that can still satisfy the Q-T1 query. This is possible only if there exists a set of helper service vectors that are causally connected to these service vectors via a dependency path such that some of the helper services (i.e. the *source services*) can be triggered using the Q-T2 query event (or the *initial event*).

We now discuss a very significant property of *abstract ALNet* that has a major impact over the proposed event-handling algorithm. We can prove, under certain assumptions, that there must be only one *abstract source node* and only one *abstract end node* for a satisfiable non-compound (i.e. simple or complex as per *DQM* model) user event.

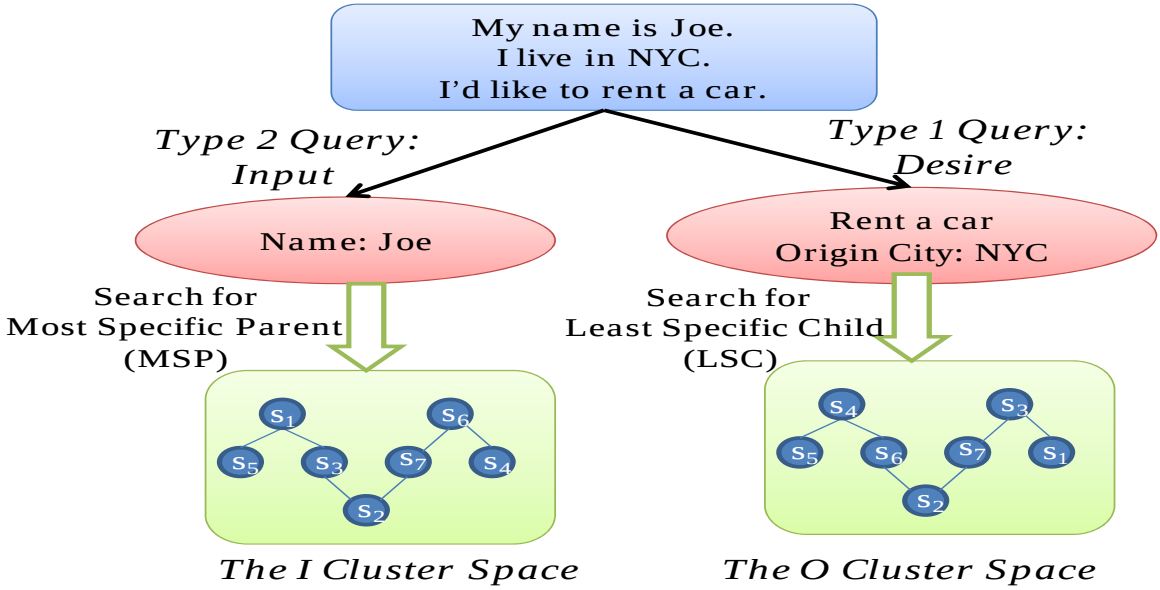


Figure 40: ALNetSniffer: Discovering Strong Solution Sets for query (Q-T1 & Q-T2)

The assumptions are that for user events that contain complex *DQM* queries the system ignores all possible alternatives where the query arguments in conjunction can be satisfied by more than one service agent in a contextually coherent manner under certain mutual negotiation. For an example, if the user event for renting a car contains a complex Q-T1 query that demands *{car_information, rent_confirmation}* then the *ALNet* system ignores the possibility that there can be two services s_1 and s_2 involved such that s_1 provides the *rent_confirmation* first and then asks s_2 to provide *car_information* of the corresponding rented car. In our model we assume that s_1 should be able to completely (i.e. member of *SSS-P1*) or partially (i.e. member of *WSS-P1*) satisfy the complex Q-T1 without undergoing any negotiation with s_2 even if the user satisfiability is not complete. We call this assumption the *strict restriction over complex Q-T1*. To prove the aforesaid discussed property formally we first prove the following lemma:

Lemma 5.1: A particular semantic description is unique to an *abstract node*.

Proof: An *abstract node* is definitionally equivalent to a *g-taxonomy*. Since a *g-taxonomy* has one root service vector (say s_r) therefore s_r *g-subsumes* all member service vectors. Therefore, the semantic description of the *abstract node* is unique if it takes the value of the description of s_r. ■

Theorem 5.2: A *target event* can be interpreted in a particular manner by at most *one abstract end node*.

Proof: Given the assumption of *strict restriction over complex Q-T1* if the desired Q-T1 of the *target event* is the set X and if X is satisfiable then there exists a service vector s_1 such that $\overset{g}{\supset}(X, s_1) \neq fail$. Hence, $\overset{g}{\supset}(X, Ancestor(s_1)) \neq fail$ $\overset{g}{\supset}(X, Descendant(s_1)) \neq fail$ is also true. Also if there exists another service vector s_2 such that $\overset{g}{\supset}(X, s_2) \neq fail$ then as *g-relation* is transitive hence,

$\overset{g}{\supset}(Ancestor(s_1), s_2) \neq fail \wedge \overset{g}{\supset}(Descendant(s_1), s_2) \neq fail$ is true. This implies that s_2 must also belong to the same abstract node that contains s_1. Hence, there exists an *abstract end node* for a given non-compound Q-T1 user event under the aforesaid assumption. As per lemma 5.1 such an *abstract end node* is unique. ■

Theorem 5.3: An *initial event* can be consumed by at most one abstract end node.

Proof: Proof logic similar to that of previous theorem except that the assumption of *strict restriction over complex Q-T1* can be lifted. This is because *initial event* contains *Q-T2* query only. ■

At this point it is to be observed that it is not guaranteed that the combined phase 1 solution set must be the *end services*. This is because there can exist a set of solution service

vectors such that there is no *dependency path* from any of the *source services*. Hence, we termed the combined phase 1 solution set as *'possible' end services*. Finding a *dependency path* is basically an *event handling process*. The problem of finding the *optimal dependency path* for a given initial event is called service composition (in the context of the proposed model *event-handling*). The *abstract ALNet* is meant only to ease the discovery of *end service nodes* and thereby improve the overall composition performance. The instance-level *ALNet* is the structure over which the real *dependency path* is discovered.

5.5.3 Dependency Path Discovery

One of the major difficulties in an *event-handling* problem is to provide a computationally efficient solution for selecting *end services* out of a set of *possible end services* (discussed in previous section). This requires discovering the dependency path from the *source services* to the *end services*. We now introduce a concept called *ALNet reachability* that serves as a computational tool for efficiently finding whether a dependency path exists between a pair of arbitrary *causal service node* and *responsive service node*. As mentioned in the related work section, *reachability* (denoted as $\Leftarrow$) is a problem of finding whether a path exists between two given vertices in a graph. In this work we propose an *ALNet encoding* technique that uses the same concept as that of *b-Encoding* for bit-based encoding of *base ontologies* discussed in chapter 3. The only difference is that unlike an ontological taxonomy an *ALNet* may not be acyclic. This is because of the special node *NN* (*Not Necessary*) that allows feedback stimuli from the responsive service to the original causal service. However, during the encoding process we convert all such reverting edges

into forward edges such that from a graph point of view all *NN* nodes are treated as parent nodes to responsive service vector nodes instead of child nodes. This transformation does not affect the problem of *dependency path* discovery since if a forward path can be computed between two service vector nodes within *ALNet* then a backward path can obviously be deduced from that. Once the global *ALNet* instance is encoded within the middleware the codes are passed on to all the local instances of each registered service vector nodes. A reachability can then be very efficiently computed by doing a subsumption testing between one service code and another service code. For an example if s_1 (say a *source service node*) has the code *0*1* and s_5 (say an *end service node*) has the code *0*1111111011* (figure 34) then since the subsumption test results in the code of s_2 hence we understand that there exists a *dependency path* from s_1 to s_2 (i.e. $s_2 \Leftarrow s_1$). However, such *reachability* computation is based on the assumption that the global *ALNet* instance topology remains constant at the point of computation. The query time complexity is O(N/W) where N is the number of service vectors in the *ALNet* instance and W is the in-memory word capacity of the computational model.

Even though query time for *ALNet reachability* is fast updating cost is still to be solved. Updating is required when a new service vector joins the *ALNet* instance or an existing service vector leaves the instance. Joining of a service vector is essentially discovering its all possible dependencies within the instance. This is done by mapping its *O-array* to the *I-cluster space* of the instance and its *I-array* to the *O-cluster space* of the instance. In the worst case this takes linear time. During joining the new service node may be: (i) leaf node, (ii) root node, (iii) parent node when it has a set of children and also a set

of parents. In the first case encoding is trivial because the new service vector just inherits existing parent node codes. In the second and third cases the global *ALNet* instance is re-encoded. Thus, the worst case insertion update time is $O(b^n)$ where *n* is the number of nodes in the ALNet and *b* is the branching factor. On the other hand deletion of service vector can also be at three levels: (i) leaf, (ii) root, and (iii) parent. However deletion of service vector does not affect the *ALNet* instance encoding in any of these three cases.

The idea of reachability can also be applied to the *abstract ALNet* as well. In a similar manner we can compute whether two *abstract service nodes* are reachable by computing the reachability between their corresponding root service vector nodes. Hence, if for all root service vectors in a given *abstract causal node* has reachability with at least one of the root service vectors in a given *abstract responsive node* then the two abstract nodes are said to have reachability. We can therefore prove using theorem 5.1 (and corollary 5.1) that if there is reachability between an *abstract source node* and an *abstract end node* then there must be reachability from all corresponding source service vector nodes to at least one end service vector node (and vice-versa) selected from these abstract nodes.

Theorem 5.4: Given abstract nodes S_i and S_{ji} if $S_j \Leftarrow S_i$ then $[\forall s_m \in S_i, \exists s_n \in S_j \ni s_n \Leftarrow s_m] \wedge [\forall s_n \in S_j, \exists s_m \in S_j \ni s_n \Leftarrow s_m]$.

Proof: Since $S_j \Leftarrow S_i$ hence, either S_j is responsive to S_i (i.e. $S_j \leftarrow S_i$) or there must exist at least another *abstract node* S_k such that $S_j \leftarrow S_k \leftarrow S_i$. In the former case proof is obvious according to theorem 5.1. In the latter case S_k is an *abstract node* that is causal to S_j and at the same time responsive to S_i. Hence, all service vectors in S_i have dependency with

all other service vectors of S_k (theorem 5.1). Also same set of service vectors in S_k has dependency with all other service vectors of S_j. Since *g-relation* (and thus, dependency) is a transitive relation hence, all services of S_i have dependency with all other services of S_j. ■

5.5.4 ALNetComposer: Event-handling Algorithm

In this section we propose our event-handling algorithm called *ALNetComposer*. The algorithm is basically a back-ward traversing greedy approach and is essentially a modified single destination shortest path algorithm. However, unlike many other popular shortest path algorithms the traversal is guided and not explorative. It is to be noted that instead of edge cost we have a node cost in our case since service vectors come with a cost of their own that can be the price, service latency, service reliability, etc. The goal is to discover a dependency path D given a user event {Q-T1, Q-T2} such that a cost function F_{cost} is minimized. In this work we do not model the cost function F_{cost} but assume it as generic. We do not include any user-defined constraint over F_{cost} as most constraint variables in the problem of service composition are QoS parameters that can be treated as decision variables within F_{cost}. We hereby provide the problem statement as follows:

Problem Statement: *Given an ALNet instance A^L and a user event consisting of the query {Q-T1, Q-T2} find a dependency path D such that $cost_D$ = Min (F_{cost}) and Q-T1 is best satisfied in terms of g-relation.*

ALNetComposer first initializes D as NULL and its corresponding cost ($cost_D$) as 0. After that it runs *ALNetSniffer-Phase 1* for getting the complete solution set (*P1_soln*). Note that this solution set contains both the WSS-P1 and SSS-P1 sets. Thus, *P1_soln* set is the

possible *abstract end node*. If *P1_soln* is empty then the algorithm terminates stating that the given user event cannot be handled. However, if it finds a solution set then it runs *ALNetSniffer-Phase 2* to find the *abstract source node P2_soln*. As per theorem 5.4, 5.2, and 5.3 for a given user event there can be only one pair of abstract source and end nodes. Thus, *ALNetComposer* utilizes this property to prune the search for *D* only to *P1_soln* and *P2_soln*. The algorithm then checks the reachability between these two solution sets. If there exists a reachability then it tries to extract the best possible end services (i.e. *SSS*) out of *P1_soln*. However if such a solution set is not found then it extracts the next best matches (i.e. *WSS*). After that the algorithm selects the service vector that has minimum cost from the extracted end service set. This end service (*End_node*) is guaranteed to have reachability with at least one service vector in *P2_soln* (theorem 5.4). Thus, the algorithm selects *End_node* as the terminating node of *D*. After that back-traversal is initiated in a way such that the next hop is an abstract causal node that has reachability to the source abstract node *P2_soln*. After that the service vector that has the lowest cumulative minimum cost is selected from the next hop. Cumulative cost is simply the addition of the current cost of D and the cost of each service vector in next hop. This service vector is then added into D. The process goes on till *P2_soln* is reached and a service vector is selected (and added into D) from that. We now provide our proposed event-handling algorithm called *ALNetComposer* in figure 41.

To improve runtime performance *ALNetComposer* follows the traversal guidelines given in table 7. It has to be noted that runtime traversal is forward traversal over the

discovered optimal dependency path by *ALNetComposer*. According to the table the *ALNetComposer* does not select any operator node as the next hope if there is an alternative

```
ALGORITHM: ALNetComposer
INPUT: ALNet Instance, {Q-T1, Q-T2} // {Q-T1, Q-T2}: user event
OUTPUT: D // D: optimal dependency path for user event
START
D = NULL;
P1_soln = ALNetSniffer(Q-T1, O-cluster space)
IF P1_soln != NULL
        P2_soln = ALNetSniffer(Q-T2, I-cluster space)
END IF
IF  P1_soln ⇐ P2_soln
        SS = findSSS(Q-T1, O-cluster space)
        IF SS = NULL
                SS = findWSS(Q-T1, O-cluster space)
        END IF
        D  = minCost(SS, D)

        WHILE next_hop ∉ P2_soln
                next_hop = selectReachableNode(O-abstract nodes, P1_soln)
                D = minCost(next_hop, D) + D
        END LOOP
        return D
END IF
ELSE
        return NULL
END IF
END
```

Figure 41: ALNetComposer: Event-Handling Algorithm

next hop. This is because operator nodes in general requires an additional decision computation when *ALNetComposer* requires to proceed forwards. If logical operators cannot be avoided then the next thing to avoid is NN operator nodes since that creates the possibility of runtime loops during event-handing. Next node that has to be avoid is post-AND. This is because during runtime post-AND forces the overall event-handling process to wait till all the responsive service vectors to the post-AND are executed. This increases overall service latency. Then comes the pre-AND node as it also incurs overall service

latency since all the causal service vectors have to be executed before the event-handling can proceed forward. Next *ALNetComposer* tries to avoid the pre/post XC operators since XC operators enforces a runtime decision to be taken regarding the selection of the service vectors connected to the XC. After the selection the system then has to stop other non-selected services from being executed.

Table 7: Runtime Traversal Optimization Guideline

Objective	Heuristic	Priority
Runtime Traversal Cost Minimization	Avoid operator nodes	1 (highest)
	Avoid NN node	2
	Avoid post-AND node	3
	Avoid pre-AND node	4
	Avoid pre/post-XC node	5
	Avoid pre/post-OR node	6

5.5.5 Situation Boundary & Event-handling Optimization

Event-handling within the *ALNet* framework can be optimized based on the proposed concept of *situation boundary* (denoted *SB*). A '*situation*' arises when a set of related events occur within an *ALNet* system such that a common user event is handled optimally. These related events, in general, can be both user feedback events as well as service events. A *situation* characterizes a particular dynamics of the *ALNet* system that is

specifically tailored for a particular event semantics. More specifically, it can be proved that for a specific kind of user event there is a fixed set of *abstract nodes* that are involved in the *event-handling* process. This fixed set is called *situation boundary* for the specific type of user event. We define *situation boundary* as follow:

Definition 5.17: If there exists a *dependency path D* that optimally handles a user event ε then a *situation boundary* for the event (denoted SB_ε) is defined as: $SB_\varepsilon = \{S_i | \forall s \in D; s \in S_i\}$ where *S* is an *abstract node* and *s* is *service vector node* in *D*. ■

In order to prove the existence of a common fixed *SB* for a given type of user events we first need a formal understanding of *event similarity* that is used to typify events.

Definition 5.18: Two events ε_i & ε_j are said to have *event similarity* (denoted as $\varepsilon_i \cong \varepsilon_j$) iff $\stackrel{O}{\sqsupseteq}(\varepsilon_i.Q\text{-}T1, \varepsilon_j.Q\text{-}T1)) \neq fail$.■

The above definition implies that if the desire component of two user events have *g-relation* (i.e. exact or plugin or subsume or sibling match) then the user events are said to be *event similar*. We now give a stronger definition of event similarity as follows:

Definition 5.19: Two events ε_i & ε_j are said to have *strong event similarity* (denoted as $\varepsilon_i \stackrel{S}{\cong} \varepsilon_j$) iff $\stackrel{O}{\sqsupseteq}(\varepsilon_i.Q\text{-}T1, \varepsilon_j.Q\text{-}T1)) \neq fail] \wedge [\stackrel{I}{\sqsupseteq}(\varepsilon_i.Q\text{-}T2, \varepsilon_j.Q\text{-}T2)) \neq fail]$.■

The stronger definition imposes an additional *g-relation* restriction over the *input* components of two user events. Based on this definition we now prove the existence of a fixed situation boundary for two event similar user events.

Theorem 5.5: Given two satisfiable user events ε_i & ε_j such that $\varepsilon_i \stackrel{S}{\cong} \varepsilon_j; SB_{\varepsilon_i} \equiv SB_{\varepsilon_j}$

Proof: For the Q-T1 components of the events to be satisfied there must exist two end service vectors (say s_i and s_j). Since $\varepsilon_i \overset{S}{\cong} \varepsilon_j$ hence, they have *g-relation* with respect to Q-T1. As *g-relation* is a transitive relation hence s_1 and s_2 are in *g-relation* to each other. Thus, s_1 and s_2 belong to the same abstract end node (say S_{end}). Also since the user events have *g-relation* with respect to Q-T2 hence, there must exist at least two services s_3 and s_4 such that they have *g-relation* with the user events Q-T2 components as well as with each other (due to transitivity of *g-relation*). Thus, s_3 and s_4 must belong to the same abstract source node (say S_{source}). Since the two user events are satisfiable hence they must have corresponding *dependency paths* - say D_i and D_j respectively. Hence, both D_i and D_j must start at S_{source} and end at S_{end}. According to theorem 5.1 (and corollary 5.1) the next abstract node hop for each of the *ALNetComposer* processes corresponding to the two user events will always be the same. Hence, SB_i is equivalent to SB_j. ■

The significance of the above theorem is that the *ALNet* middleware stores *SB* information in its knowledge base for every new event type so that if it observes similar event in future it can pull up the *SB* and starts off *ALNetComposer* for the newly observed event to discover the optimal dependency path just within the selected *SB*. The same principle can also be applied to events that are weakly event similar. However, for this case corresponding user event *SB*s will have overlap and not equivalent. *SB overlap* is a phenomenon when two *SB* sets have an intersection but the intersection is not equivalent to their union. It is to be observed that the final *abstract end node* will be same for the two *SBs* and thus, is a minimum *SB overlap* possible. *SB overlap* is currently outside the scope of this

work. The following algorithm, called *SBTraveller*, optimizes the *ALNetComposer* algorithm.

```
ALGORITHM: SBTraveller
INPUT: ALNet Instance, {Q-T1, Q-T2} // {Q-T1, Q-T2}: user event
OUTPUT: D // D: optimal dependency path for user event
START
D = NULL;
Event_type = getStrongEventMatch({Q-T1, Q-T2}, KBase) //KBase: Knowledge base
IF Event_type == NULL
        D = ALNetComposer(ALNet Instance, {Q-T1, Q-T2})
        return D
END IF
ELSE
        Current_SB = getSB(Event_type, KBase)
        P1_soln = getAbsractEndNode(Current_SB)
        D = next_hop = minCost(P1_soln, D)
        P2_soln = getAbsractSourceNode(Current_SB)

        WHILE next_hop ∉ P2_soln
                next_hop = P1_soln.parent
                P1_soln = P1_soln.parent
                D = minCost(next_hop, D) + D
        END LOOP
        return D
END IF
END
```

Figure 42: *SBTraveller*: An optimized *ALNetComposer* algorithm

5.6 Results: Service Discovery Accuracy

One of the primary objectives of the evaluation discussed in this section is to evaluate the precision/recall of *ALNetSniffer* algorithm when a set of *DQM* query based user events. This approach resolves the inadequacy of domain based accuracy measure by strictly restricting the accuracy evaluation over the set of services that are retrieved as functionally similar to a given complex *DQM* query. Moreover, this approach neatly evaluates the service discovery performance of the *ALNet* framework as a whole. We first define *query-precision* and *query-recall* as two measures for understanding the accuracy of a discovery

process and hence, the goodness of a cluster space generated by a given service category learning algorithm.

Definition 5.20: *Query-Precision* with respect to a given query Q is defined as the ratio of the number of relevant services that are retrieved (say $n_{rel,Q}$) when Q is mapped over the cluster space generated by a service category learning algorithm L vs. the total number of retrieved services ($N_{ret,Q}$) for Q. Numerically this means: $Pr(Q) = n_{rel,Q} / N_{ret,Q}$. ■

Definition 5.21: *Query-Recall* with respect to a given query Q is defined as the ratio of the number of relevant services that are retrieved (say $N_{rel,Q}$) when Q is mapped over the cluster space generated by a service category learning algorithm L vs. the total number of relevant services ($N_{rel,Q}$) for Q. Numerically this means: $Re(Q) = n_{rel,Q} / N_{rel,Q}$. ■

To evaluate the *query-precision* and *query-recall* of STC we used the query set given in OWLS-TC v2 dataset. The query set contains 29 queries each accompanied by its corresponding expert-evaluated set of relevant services. We pre-processed the queries by converting them into their corresponding query type (i.e. simple, complex, or compound) as per the proposed *DQM* query modeling in chapter 3. We observed that all the queries were either simple or complex in type. We first calculated the *average 11 point precision over recall* for each of the 29 queries. This accuracy measure helps us to evaluate the precision as well as the recall in an integrated manner. The underlying idea is to rank the retrieved services in order of relevancy to a given query and then to analyze each of the ranked services one by one by measuring the precision observed till that rank and the recall with respect to the given relevant set of services for that query. The measure is formally known as *precision@r*. We define it as follows:

Definition 5.22: *(Precision@r)* Given a query Q and its set R_Q of relevant services ($N_{rel,Q}$) if a service discovery process over a cluster space generated by a service category learning algorithm L retrieves an ordered set of services $R_{ret,Q}$ (cardinality, say, $N_{ret,Q}$) where the order is in decreasing sequence of relevancy of member services to Q then the *precision @ r* (denoted as *Pr(Q, r)*) is defined as the ratio of number of services in subset $R^r_{ret,Q} = \{s_1, s_2, ..., s_r\}$ that are members of R_Q (say, $N^r_{rel,Q}$) over the total number of services in $R^r_{ret,Q}$ (i.e. r). ■

The *precision @ r* is numerically calculated as: $Pr(Q,r) = N^r_{rel,Q} / r$. The corresponding *recall @ r* is numerically calculated as: $Re(Q, r) = N^r_{rel,Q} / N_{rel,Q}$. Note here that r is a positive integer such that $r = [1, N_{ret,Q}]$. r is often called the *cut-off* point. To give an example, let us assume that the set $R_Q = \{s_2, s_7, s_1, s_4, s_5, s_{12}, s_{74}, s_{40}, s_{42}, s_{34}\}$ where R_Q containing the relevant services for a query Q has been given. Thus, there are 10 services that are relevant to the query Q, as determined by the domain experts. Suppose a service discovery algorithm returns for the query Q the ordered set $R_{ret,Q} = \{s_{34}, s_{22}, s_{40}, s_{98}, s_{100}, s_4, s_{500}, s_{600}, s_{17}, s_{12}, s_{21}, s_{23}, s_{33}, s_{51}, s_{74}\}$. We find that the service s_{34} which is ranked number 1 is relevant. Since this service corresponds to 10% of all the relevant services in the set R, we say that we have a precision of 100% at 10% recall. The service s_{40} which is ranked as number 3 is the next relevant service. At this point, we say that we have a precision of roughly 66% (2 Service out of 3 are relevant) at 20% recall (two of the 10 relevant services have been seen). If we move further down the list, precision of 50% at 30% recall; precision of 40% at 40% recall; precision of 30% at 50% recall; 0% precision at >50% recall. Since different queries may generate different *cut-off* points (based on the cardinality of the $R_{ret,Q}$

set) the standard way of evaluating the discovery performance over a set of queries is to interpolate the *cut-off* points into 11-points for each query and then averaging the *precision@r* over the number of queries. We define *average 11-point interpolated precision@r* as:

Definition 5.23: The *interpolated precision @ r* (denoted as *I_Pr(Q,r)*) at a certain recall level *r = [1, 11]* is defined as the highest precision found for any recall level $r' \geq r$ $I\_Pr(Q,r) = Max_{r' \geq r} Pr(Q,r')$. ■

Definition 5.24: The *average 11-point interpolated precision @ r* (denoted as *I_Pr(r)*) at a certain recall level *r = [1, 11]* is defined as the averaged interpolated precision @ r over the total number of queries that have been mapped over the cluster space by the discovery algorithm:

$$I\_Pr(r) = \frac{\sum_{Q=1}^{N_Q} I\_Pr(Q,r)}{N_Q} \blacksquare$$

So, for the previous example, at recall levels 0%, 10%, 20%, 30%: the interpolated precision is equal to 33.3%. At recall levels 40%, 50%, and 60%, the interpolated precision is 25% (which is the precision at the recall level 66.6%). At recall levels 70%, 80%, 90%, and 100%, the interpolated precision is 20% (which is the precision at recall level 100%). We first observed the performance of the proposed discovery algorithm over the *O-cluster space* formed by STC for each of the 29 Q-T1 queries. This was to evaluate the accuracy performance in the phase 1 service discovery process. To evaluate we calculated the *mean interpolated precision @ r* over the 11 points for each of the queries (figure 43). In general for most queries (except for 10 queries) the phase 1 discovery process obtained an average

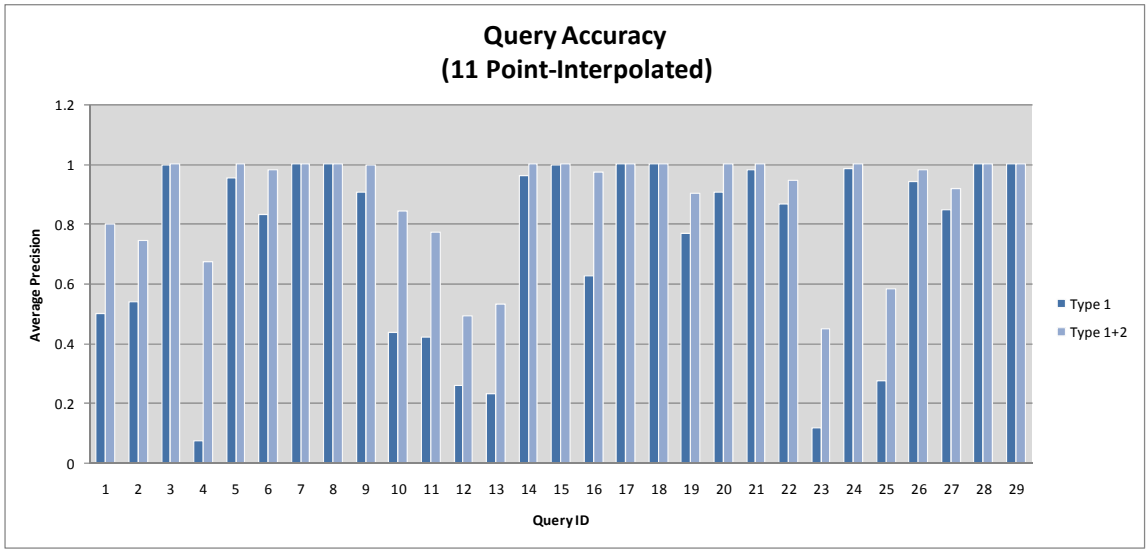


Figure 43: Query Accuracy (MIP): Phase 1 & Phase 2 *ALNetSniffer*

interpolated precision (MIP) near to 0.8. We next repeated the experiment by completing the 2-phase discovery process with the inclusion of phase 2 discovery (figure 43). We observed a significant increase in the overall average interpolated precision after the 2nd phase was done. This was because a considerable number of services that were identified as functionally similar to the desire part of the Q-T1 components of the queries during phase 1 where eliminated out as services that cannot be dynamically called either directly by the query (by providing the required input in the Q-T2 component) or indirectly by some other services that can be called themselves. This phenomenon supports the fact that discovery has to be carried as a 2-phase process.

We then compared the *mean average interpolated precision* (i.e. *average interpolated precision over 11 points* when averaged over all the queries) with that of 6 other prominent service discovery algorithms (figure 44). All these 6 algorithms (except Woogle) are implemented over a dataset of 391 web services collected from 11 different

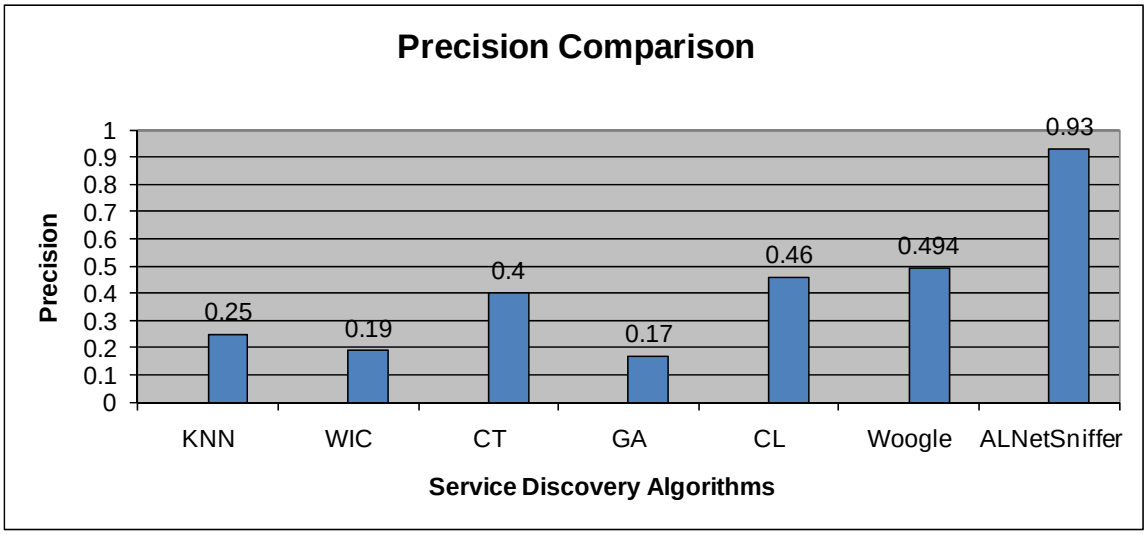


Figure 44: Comparative Analysis of Mean Interpolated Accuracy

domains (source: SALCentral.org). Out of these 6 algorithms WIC (Word-IC algorithm [177]), Woogle [171] and CL (Complete Link algorithm [163]) are based on hierarchical clustering:

- ***WIC***: WIC uses a distance measure that is based on common words between two services and relies on a global cluster quality function [177] that acts as the merging condition for two clusters.
- ***Woogle***: In comparison, Woogle [171] is implemented over a dataset of 1574 web services over 8 domains Woogle is a merge-split based hierarchical clustering algorithm where, unlike most other service category learning algorithms, a cluster is a *concept* (or set of similar terms) instead of a set of similar services. The distance measure used to form such *concept* is based on overall association strength of the member terms (called *cohesion*). The merging condition implies that all terms within a cluster have to be representative terms (called *kernels*) – that is, terms that have high cohesion with at least

half of the other member *kernel* terms within a newly formed cluster. Hence, to satisfy this condition splitting may be necessary. Two services are grouped together based on the *concept match* (i.e. whether two Input/Output parameter terms within the web service descriptions belong to the same *concept*). Matching between the parameters (*Input* and *Output*) is carried over separately from each other. However, the overall similarity score is integrated.

- ***CL***: CL is a variation of the hierarchical agglomerative hierarchical clustering algorithm.
- ***GA***: The GA (Group Average algorithm) is an improvisation over CL.

CT is essentially partitional clustering algorithm as per follows:

- ***CT***: In the case of CT (Common Term algorithm [163]) the distance measure is cosine similarity. The iteration process is similar to k-means except that the centroid that is shared by all the clusters is selected for the next iteration (unlike k-means where the cluster members are averaged to get the new centroid).

In comparison to all these 6 algorithms we found that *ALNetSniffer* has a significant edge in terms of *mean average interpolated precision* over all of these. However, it is important to note over here that the dataset used by *STC* (i.e. OWLS-TC v2) is semantic in its representation (web service description language used is OWL-S) while all the others have used datasets that are based on the standard syntactic web service description language WSDL. This greatly enhances the precision level of the algorithm in comparison to others that used statistical learning techniques. Another reason for this significant improvement is because *ALNetSniffer* utilizes *STC* that generates functionally similar taxonomies based on semantic similarity (*g- subsumption*). Hence, theoretically (as has been proved in chapter 4)

its accuracy is sound and complete. However, in reality this promise is based upon the assumption that the service providers have to describe their services semantically as precise and as complete as possible. Since this assumption is not true in many cases hence, we observe that although *ALNetSniffer* scores well compared to all the 6 algorithms still *STC* fails to achieve the perfect accuracy. When we compared the F-score of *ALNetSniffer* with 5 of the previous 6 algorithms (Woogle has been excluded due to lack of data) we found that it outperforms all the 5 algorithms again. We included the results of two supervised learning algorithms - *Naive Bayes* and *SVM (ensemble)* within this study. We observed that the F-1 score did not come out good for all the 5 unsupervised learning algorithms (figure 45). This is because their individual recall was not good enough when compared to their precision. In comparison to these 5 unsupervised algorithms the supervised learning algorithms fared well because of a prior training phase.

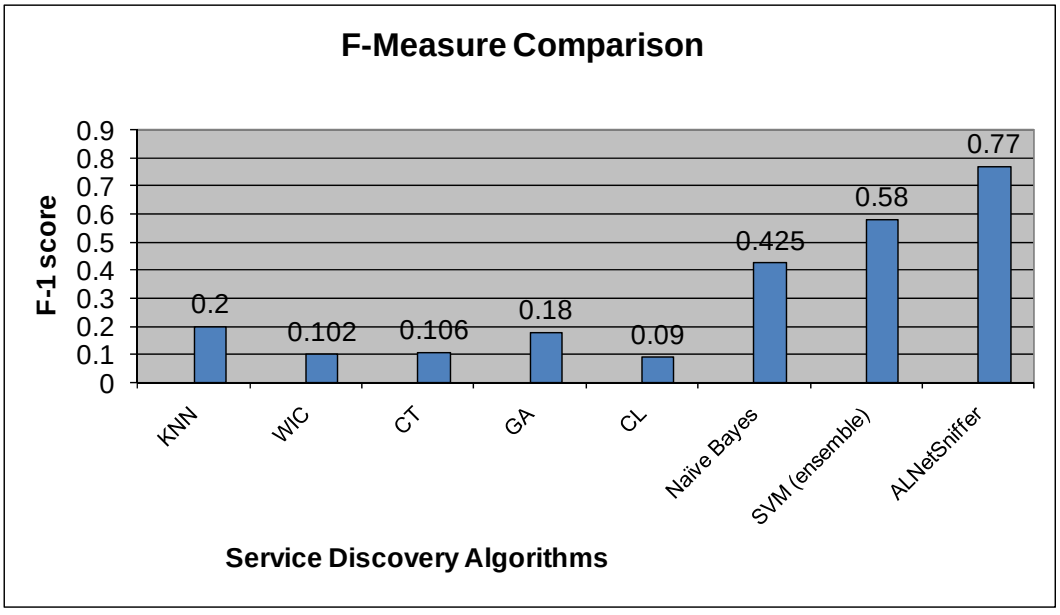


Figure 45: Comparative Analysis of F-measure

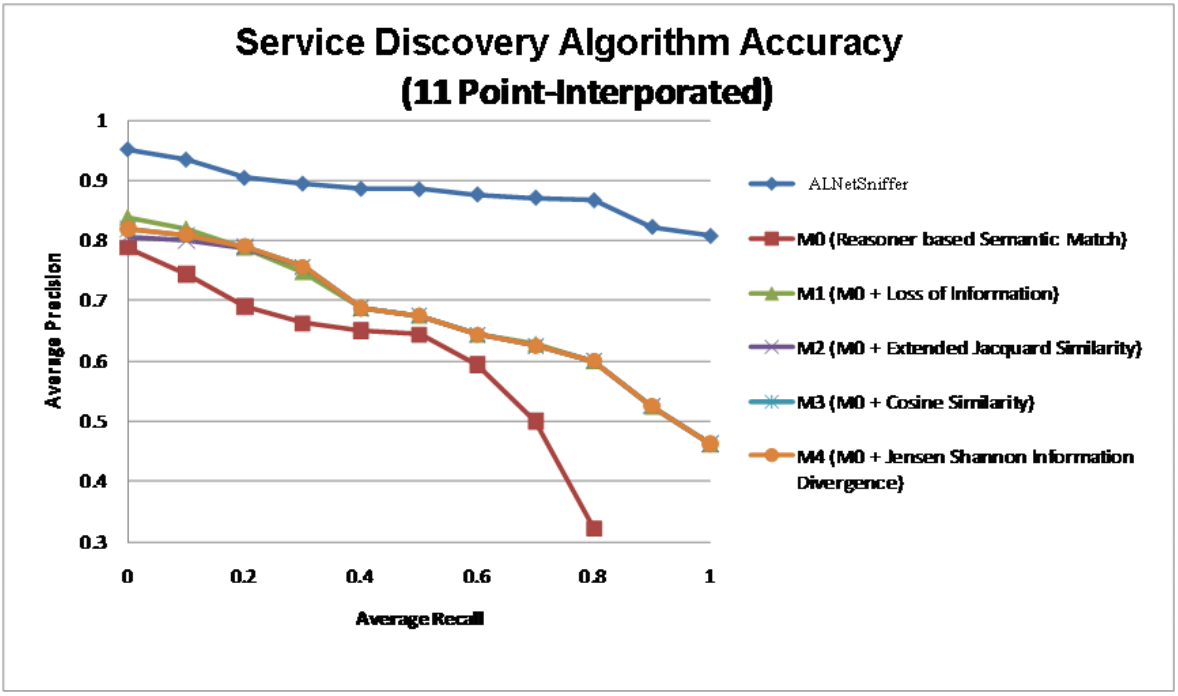


Figure 46: Comparative Analysis of Query Processing Accuracy (Precision vs. Recall)

We also compared the *average 11 point interpolated precision @ r* (i.e. *I_Pr(r)* defined previously) versus the *average recall @ r* of *ALNetSniffer* with that of 5 different service discovery techniques proposed in the OWL-S MX [147] (figure 46). The objective behind this study was to understand how our proposed *g-subsumption* matchmaking fared when compared to other matchmaking techniques that have used the same OWL-S TC v2 dataset as we did. M0-M4 are the different types of query match algorithms compared by OWL-S MX. M0 is a pure Description Logics [39] based matching algorithm that considers only the semantic definitions of the *Input/Output* parameter terms. M1 through M4 are hybrid matchmaking techniques that use both semantic definitions of parameters as well as tokens recovered from textual descriptions of services. M1 makes use of loss of information measure (LOI), M2 uses extended Jacquard similarity coefficient [203], M3 uses the cosine

similarity value [204], M4 uses the Jensen-Shannon information divergence based similarity value [205]. We found that *ALNetSniffer* had a significant improvement over all these matchmaking measures even when textual descriptions were not included into the service feature by *ALNetSniffer* (unlike M1-M4). The reason for *ALNetSniffer* having much better accuracy performance in comparison to M0 is that although both of them are purely based on subsumption matching of parameters yet the clustering technique that uses M0 is based on ad-hoc comparison with the innate assumption that clusters are mutually disjoint. This falsely excludes services that may have a subsumption match with member services in multiple clusters. Also the case of sibling matching (e.g., *car rental* and *bus rental*) is not accounted for in M) matching. This again falsely splits services into separate disjoint clusters. Moreover, M0 being based on the Paolucci order of matching allows false inclusion of services within clusters as strong matches. This is because of higher universal preference of *plugin* match over *subsume* match that assumes that the match strength order is preserved for both the *Input* and the *Output* features (see chapter 3).

To conclude our study on the accuracy performance of *ALNetSniffer* over *ALNet* instances we tried to understand the goodness of the clusters generated by *STC* integrated within *ALNetSniffer* in terms of *cluster entropy* when the service set to be clustered was confined to the relevant sets given for each query within the OWLS-TC v2 dataset (figure 47). The objective was to observe the instability caused to each of the relevant sets by *STC*. For measuring the instability we chose to compute the entropy of the relevant sets conditioned on the *O-clusters* that were formed when each of the relevant sets were fed as a test sample space to *ALNetSniffer*.

The entropy is computed based on the following equation:

$$H(R_q) = -\sum_{i}^{M} \frac{N_{C_i}}{N_{R_q}} \times \log \frac{N_{C_i}}{N_{R_q}}$$

where $H(R_q)$ stands for the entropy of resultant set R_q of query q, M is the total number of clusters (i.e. splits) formed by *STC* incorporated in *ALNetSniffer*, N_C is the number of services in cluster C common to R_q, and N_{Rq} is the total number of services in the relevant set. Lower is the entropy the better is our algorithm with respect to the relevant set. Ideally, a clustering algorithm should be able to reproduce the entire relevant set within a single cluster without splitting it. In that case the entropy for the relevant set is 0. If the algorithm splits up the result set into clusters with single member samples then it performs worst. When we tested the entropy effect of our algorithm on the relevant result sets we observed average entropy of 0.19551 over 29 relevant sets.

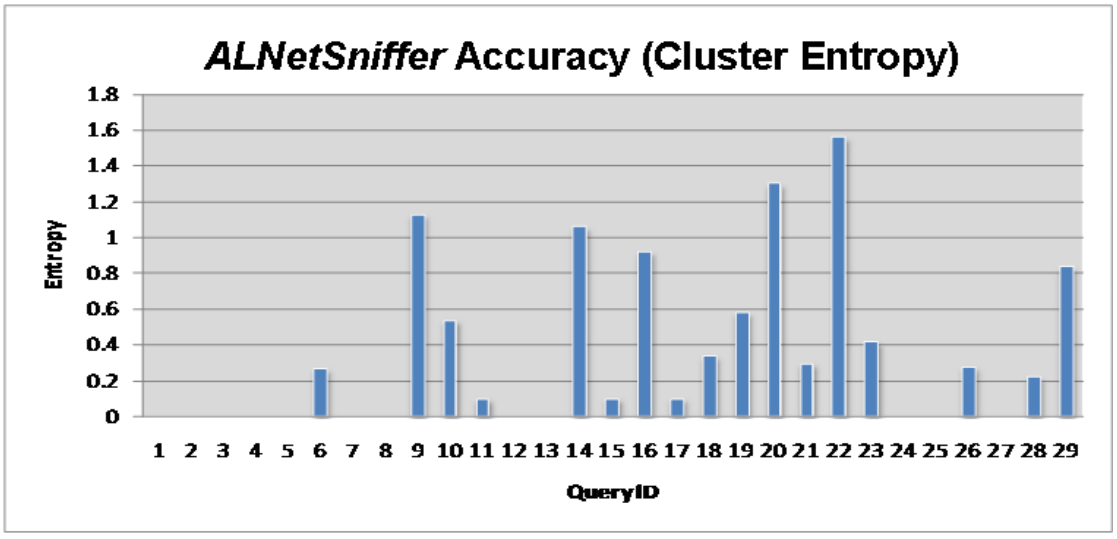


Figure 47: Query Processing Accuracy of *ALNetSniffer* (in terms of Entropy)

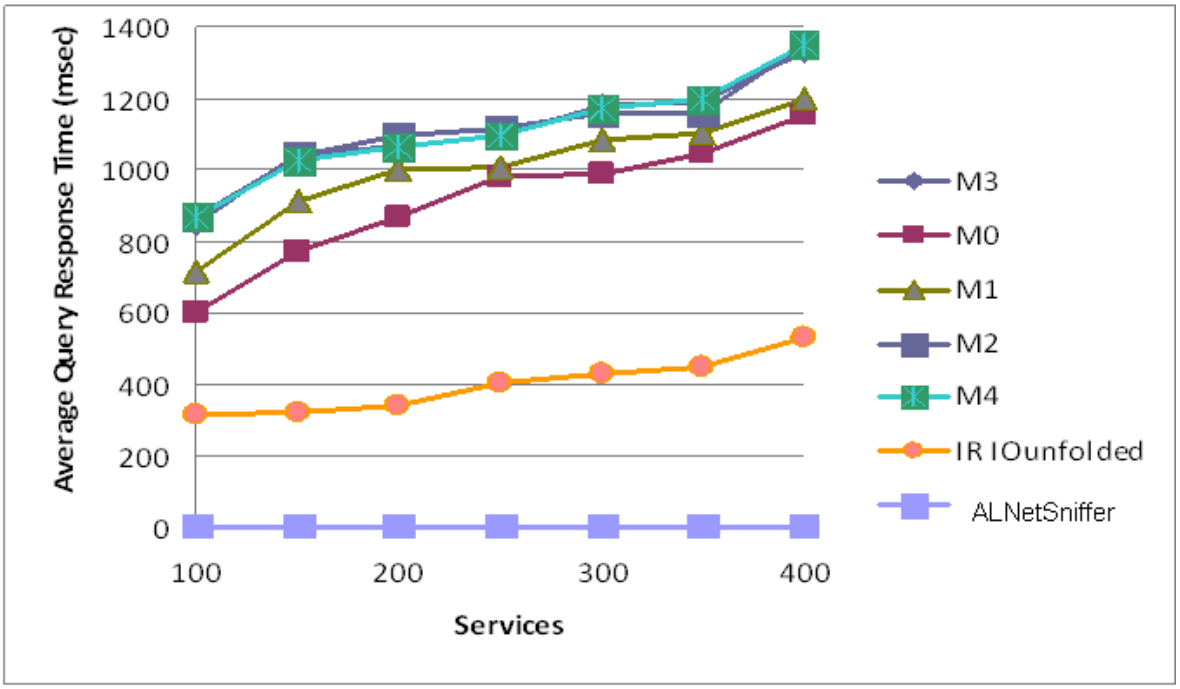


Figure 48: Comparative Analysis of Average Query Response Time

5.7 Results: Query Processing

We also studied the efficiency of the *ALNet* framework in terms of query processing overhead. We observed the average query response time performance over the OWL-S TC sample space. A significant improvement in the average query response time of *ALNetSniffer* was recorded when compared to logical-reasoning based web service retrieval (such as OWLS-M0) and other IR based hybrid models of retrieval (such as OWLS-M1, M2, M3, M4) (figure 48). This is primarily because of two reasons: (a) the compact *DQM* representation of service vector features as *g-arrays* and (b) the absence of DL-reasoning based subsumption computation (by using *DLEncoding* based *g-subsumption* algorithm).

5.8 Results: Event-handling

The experimental setup for evaluating event-handling over *ALNet* instances has been designed as a simulation platform where the run-time performance of the *SBTraveller* algorithm can be tested. The experimental platform was a machine with CPU cycle of 1.4 GHz and RAM of 2 GB. The development platform was NetBeans IDE 6.0.1. The evaluation criteria of the *SBTraveller* are based upon a set of given *ALNet* instances. For that we have developed an *ALNet Generator* that randomly produces such ALNet instances based on two test parameters: (i) number of service vector nodes and (ii) number of ALNet logical function nodes. We generated a set of abstract ALNet instances of sizes that varied between 80 – 700 abstract service nodes. The *SBGenerator* also assigns randomly generated service vector features (*Input* and *Output* parameters) from a pre-defined CAOFES instance to each of the service nodes in the generated *ALNet* instances. We used a set of 20 random

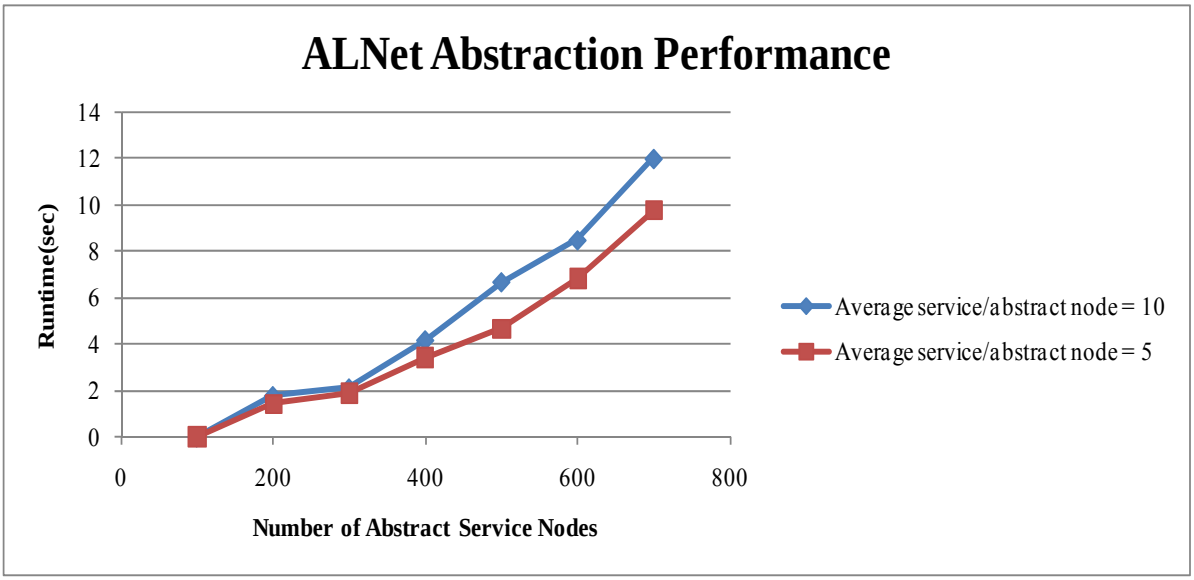


Figure 49: ALNet Abstraction over *Abstract ALNet* Instances

test events for the *SBTraveller* algorithm to be evaluated. Hence, we developed an *Event Generator* to randomly produce events (and corresponding target events) that have arbitrarily assigned event semantics. We then mapped those events (and corresponding target events) to chosen set of services that can act as source (and correspondingly end) services. The objective of the evaluation was to:

- Test the abstraction performance of *ALNetComposer* over the *ALNet* instances in order to get the corresponding abstract *ALNet* instances.
- Evaluate the *SBTraveller* runtime performance over the *abstract ALNet* instances.

For the first objective we tested the abstraction by first keeping an average of 5 service nodes per abstract node and then an average of 10 service nodes per abstract node during the random network generation process. We did not observe significant difference in the abstraction overhead when the abstract node size was doubled for the generated *abstract ALNet* instances (figure 49). Hence, the result supports the runtime dynamic creation of *abstract ALNet* instances by *ALNetComposer* during event handling.

For the second objective we observed a near-quadratic runtime behavior of *SBTraveller* versus the size of the *SB*s so formed for *abstract ALNet* instances (figure 50). As the number of abstract service nodes increases (with an average of 50 service vector nodes per abstract node) we found an increase in the logical nodes within the instances. Because of that the number of alternative path for a particular composition increased with the overall *abstract ALNet* instance size. This resulted in more computation time for deciding the best dependency path during the event-handling process. Moreover, we also

need to account for the time consumed to apply the heuristics whenever such logical nodes are encountered.

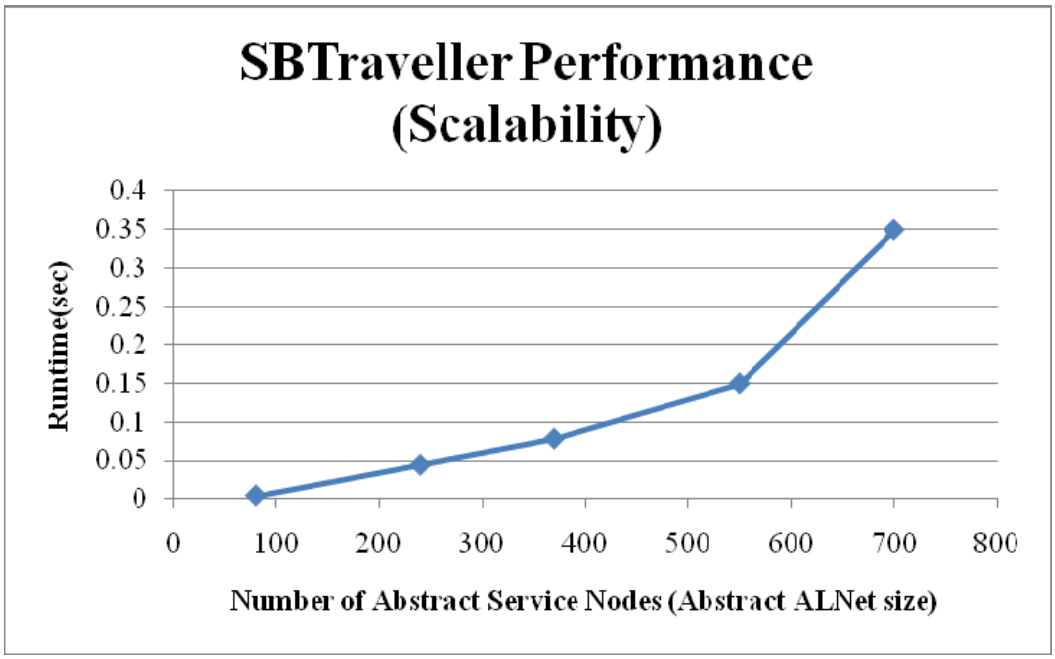


Figure 50: SBTraveller Runtime Performance (in terms of System Scalability)

We now provide a comparison of the same result with respect to number of logical nodes encountered within the SBs in figure 51. The result shows the performance of the *SBTraveller* in terms of the complexity of the *SB*s that were formed in the experiment. We observed a maximum of 0.35 secs for event-handling for the most complex SB of 90 operator nodes. Figure 52 shows the distribution of the different type of operator nodes that were involved in each of the *SB*s formed. We observe that for the most complex SB of 90 operator nodes we have the most number of NN nodes as compared to other nodes and an equal number of AND nodes (pre + post). Since these two types of nodes contribute most to

the SB complexity therefore *SBTraveller*_does a reasonable efficient job in computing the optimal dependency path in around 0.35 secs.

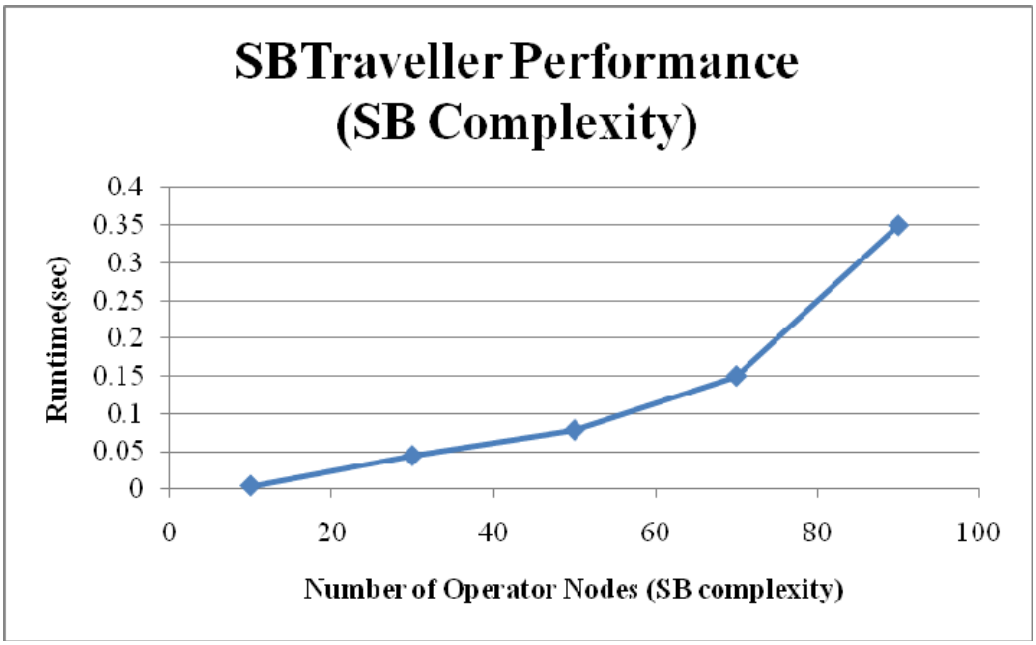


Figure 51: SBTraveller Runtime Performance (in terms of System Complexity)

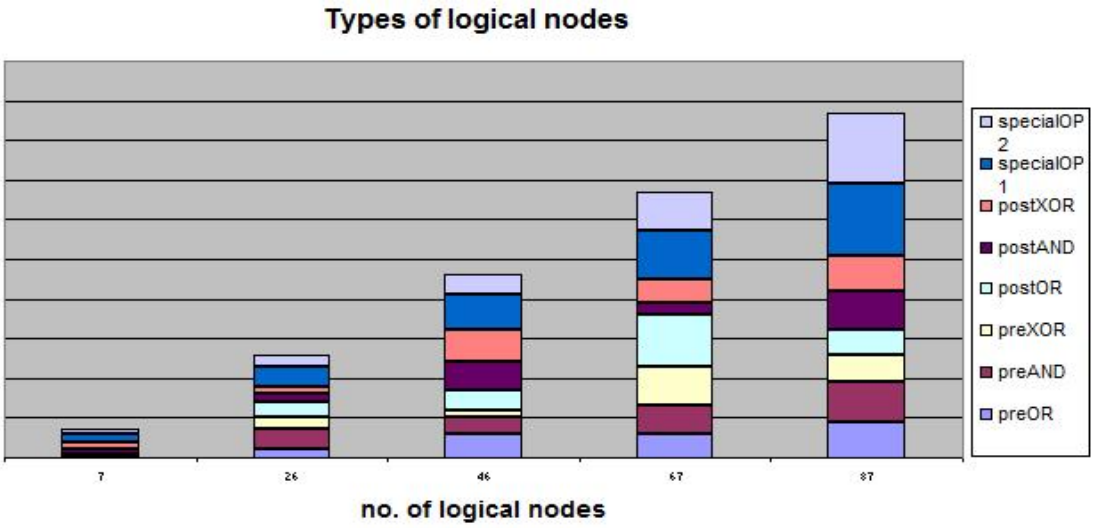


Figure 52: Operator Node Distribution w.r.t. Number of Logical Nodes in SBs

We also observed the relation between the SB size and the runtime behavior of *SBTraveller*. We see in figure 53 that for a maximum length of 176 nodes (with 90 operator nodes and 86 service vector nodes) the *SBTraveller* on an average handles an event within 0.35 secs.

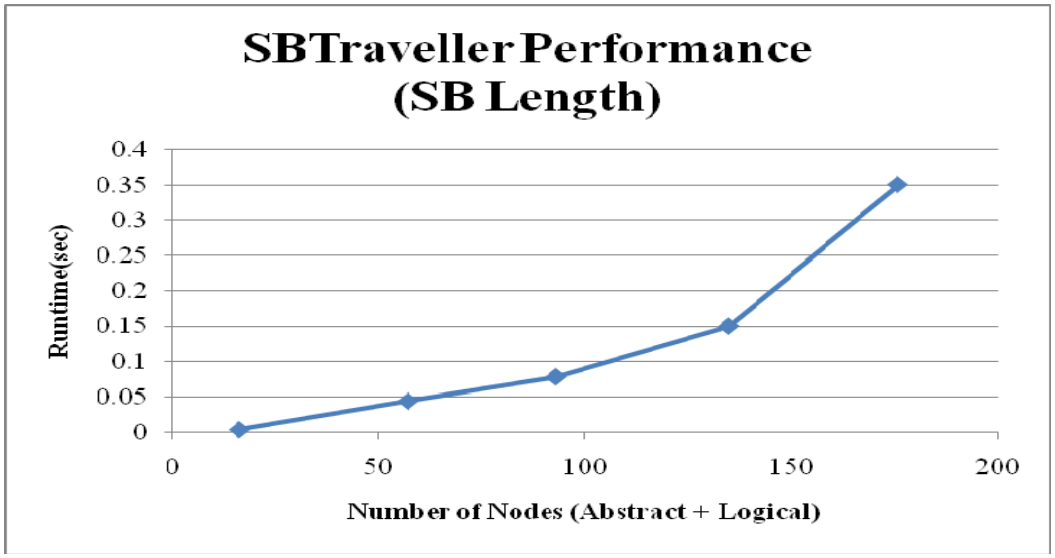


Figure 53: SBTraveller Runtime Performance (in terms of SB Length)

5.9 Conclusion & Discussion

As mentioned in the related work section ALNet based *event-handling* approach of solving the service composition problem has some striking similarity with the work of Lang & Su [198]. Even though the overall problem objective is same as compared to our proposal yet there are some major differences that have direct relationship to the performance of composition as such. Structurally ALNet is different from SDG in that the nodes of ALNet can represent only services (and not data) and logical operators. The data (input and output) is implicitly defined as feature in the service nodes that are in turn vectors. This reduces the

space complexity (as data nodes can be very large in number) as well as search time (as the network can become very large). Also data nodes in SDG are by default considered as OR nodes as more than one service can output the data while service nodes are by default AND nodes as at least one data node is required as input to the service node. However, the causal relation (or dependency) between two services are not explicitly defined in SDG but rather via data nodes making causal inference indirect. This clearly affects the composition time as for composing two services an extra hop to the data node is required. Also the more complicated relation of multiple services being causal to a service is computationally intensive to infer. Just by hopping to a data node having multiple service nodes parents does not bring us to conclusion that there exists a causal relation with the child service node of the data node to the parent service nodes via an OR operator because it may very well be that one of the parent service nodes has yet another output data node link that in turn is the input parent to the same child service node.

The absence of the NN operator (as defined in ALNet) in SDG is justified by the inclusion of cycles among data nodes where a particular data node can have another data node as a parent as well as a child. Such constructs are meant to deal with cases where an input may have to be updated or refined or repeated as a confirmation in order to be taken as an input by the same service node. However, due to lack of any explicit label over such edges it becomes impossible for a composer to differentiate whether an edge is a repetition or whether it needs a repetition in future. Thus, an indefinite loop is possible during the composition.

Another important difference is that SDGs are constructed on-demand basis where the user query is first analyzed to understand the required service categories and then an SDG involving the services within the selected categories is chosen for search. While on one hand this approach improvises over the search space but on the other hand it has to incur huge overhead by constructing new SDGs over relatively short period of time with new kinds of queries coming in. In this respect the abstraction technique of ALNet and *SB* learning can be very useful as one of the advantages of abstraction is that no two services having same functionality may belong to different abstract nodes. As the composition primarily takes place over the abstract ALNet hence, this property restricts the search space of the composition within fixed SBs without having the requirement to construct the network on query basis.

In order to maintain the completeness of the composition algorithm in the case of SDG intermediate services are added to a failed composition such that the inclusion can bring the composer to the final desired state. This approach is an added overhead since in ALNet the intermediate services are by default part of the composition and is not included unless required since the problem objective in the case of ALNet is path optimization.

When we compare SDG+ [200] (a modification to the SDG construct) with ALNet we see that most of the disadvantages of SDG still remain within the SDG+ data structure. However, there is a certain degree of reduction in unnecessary levels within the service network due to the exclusion of multi-level partitive hierarchy of data nodes in SDG+. This surely improves the searching time for a solution path but if compared to ALNet can still be recognized as a space (and hence, time) overhead. Also, the storing of SDG+ constructs for

future use is query centric and hence, the likelihood of reusability of a particular SDG+ construct purely depends upon the likelihood of its corresponding query to be triggered. Also there seems to be no obvious way in handling new queries that are similar to previously seen queries in some sense (specialization, generalization, or partly similar). In such cases SDG+ constructs needs to be created from scratch. Thus, we see that not only this model potentially introduces significantly large storage of near-redundant SDG+ constructs but it also involves the additional computation of SDG+ constructs whenever a new query is seen. In comparison *ALNet* being a query independent service network, where new services can be added or deleted with minor changes to the existing topology, we find that such problems can be eliminated without much additional overhead.

CHAPTER 6

SMARTSPACE: DISTRIBUTED MULTI-AGENT BASED EVENT-HANDLING

6.1 Introduction

In the previous chapter discussions on service discovery and composition were limited within the scope of middleware-based centralized SOA platform. However, middleware based service composition suffers from some major drawbacks that are innate in all centralized systems: (i) single point of failure, (ii) high network overload leading to service delay, (iii) impractical assumption that the system is closed and static. The proposed *ALNet* framework in the previous chapter solves some of the aforesaid drawbacks even though it is primarily middleware based. For an example, the platform does not have to assume that the system is closed and static. New services can enter the system and old services can leave during the composition process itself. This is because the *SBTraveller* algorithm primarily finds an optimal *SB* and then suggests an optimal *dependency path*. However, during runtime (which is a forward traversal over the SB), service vectors call each other based on their current availability and reachability to the current best end service vector. In other words, the runtime traversal is not governed by the *ALNet* middleware. Also even though service discovery and composition is separate in the *ALNet* platform yet the discovery is primarily of the pair of abstract source and end service nodes. Owing to some of the properties that have been proven in chapter 6 we see that for a given user event there can be only one such pair of abstract nodes. Hence, all dynamic additions and deletions of services that are pertinent to a particular event-handling process must occur within a

discovered *SB*. In this way the proposed *ALNet* platform is resilient to dynamic changes in the system.

However, there is significant room for improvement in many other aspects that will be discussed in this chapter. This has led to some very important research studies on distributed SOA platforms. Distributed SOA-based systems are very different to middleware-based SOA systems in several ways. The assumption of a centralized coordinator (and decision maker) is relaxed. In a distributed setup all service hosting agents are peer to each other. The discovery and composition process, in general, cannot be seen as two separate processes unlike centralized approaches. This is because the system state is non-deterministic and open to unseen values. Hence, discovery of services has to account for the current system state so as to find an optimal composition. Most of the research works on distributed service composition can be classified into two groups: (i) message-based [206 - 214] and (ii) agent-based [215 - 237].

In message-based models the network layout over which service discovery and composition is carried is usually P2P. Either service requests are flooded into the network for discovery or service advertisements are flooded over the network. Sometimes a hybrid model has been proposed where both advertisements and requests are flooded for discovery [211]. Message-based systems are collaborations that are governed by a fixed set of protocols. Collaborations are basically a choreography where each service hosting node's role within the P2P overlay has been fixed based on a fixed set of composition rules. In agent-based approaches the underlying system is modeled as a distributed multi-agent based system. In such a model services are transformed into software agents that are cooperative in

nature and are adaptive to dynamic system changes. Software agents proactively engage into a deal-based collaboration (rather than a centralized orchestration) based on their interpretation of the current system state. Agents communicate with each other through messages that are mostly in the request mode (i.e. an agent cannot control the behavior of other agents). In contrast to message-based systems there is no strict guideline or protocols that the agents must follow. Agent behaviors are very independent and each agent is the sole master of its set of behaviors. The choice of a particular behavior can be governed by very complex and abstract rules. Sometimes it may also be governed by an optimization function (basically the consumer utility). Thus, agent-based service composition is a much more flexible approach and is specifically useful in complex heterogeneous systems as compared to centralized middleware based approaches.

In this chapter we propose *SMARTSPACE* - a novel multi-agent based distributed platform for service composition. *SMARTSPACE* is built on top of the JADE multi-agent platform [125]. JADE is a FIPA (Foundation for Intelligent Physical Agents) compliant agent development toolkit that provides efficient support for agent-based simulation environment. Agents within this framework communicate over either an HTTP-based MTP (Message Transport Protocol) or a CORBA (Common Object Request Broker Architecture) IIOP-based MTP. We chose to work over HTTP-based MTP. More detailed information regarding JADE can be found in chapter 2. In SMARTSPACE all computing entities are transformed into agents. This includes services and user requests (in *DQM* format). Hence, there are a set of agents, called *service agents* (denoted *SA*) that embody and execute the services as and when needed. Such services are essentially agent behaviors that are

semantically described OWL-S specifications and formatted as *g-arrays* (see chapter 3). Similarly another set of agents, called *user agents* (*UA*), are generated for each user event. The user agents embody both the Q-T1 and the Q-T2 component of the user event. During an event-handling process the user agent and the service agents enter into a *deal* with the help of intermediary helper agents provided by the SMARTSPACE platform. The goal of the user agent is to maximize the user utility (cheap, best desire match, low service latency, etc) by: (a) providing as much information as possible to the helper agents, and (b) making and confirming the best deals possible at a particular instant of time. The goal of the service agents, on the other hand, is to maximize the service provider's utility (i.e. profit, reputation, etc) by: (a) outbidding other service agents for a given user agent deal, and (b) providing best user satisfaction. The dynamics is primarily that of a game with the restriction of only fair moves by any particular agent. The helper agents, as will be discussed, are unbiased to both the user agents and the service agents. In other words, there is no scope for false representation of information or rogue behavior by impeding other agents' goals.

The principal objective of *SMARTSPACE* is to provide a flexible platform where service composition can be done in an efficient way considering the uncertain dynamics of the underlying system without trading off with the problem of single-point-of-failure and incurring network overhead. Also the computational overhead of discovering an optimal dependency path is distributed over all participating agents without relying on a middleware. The chapter first introduces some of the important works that has been done in the problem of distributed service composition. It then reveals some of the major limitations of centralized service composition in general and the proposed *ALNet* platform in particular.

After that the chapter discusses the *SMARTSPACE* architecture in detail laying out all the different agent roles. Having laid the architectural foundation it then proposes *SmartCluster* – a modified distributed version of *STC* algorithm that helps to discover service agents in an efficient way. The chapter then proposes *SmartDeal* – a novel distributed service composition algorithm that achieves a near-optimal composition that is better than the previously proposed *SBTraveller* algorithm.

6.2 Related Work

There has been significant research in the area of agent-based service composition. One of the earlier works in this direction is the Web Service Modeling Framework (WSMF) proposed in [48]. Agent based service composition is mostly built on agent-based brokerage [229], service matchmaking [133, 230], and service coordination [231]. In [215 - 216] it has been pointed out that SOAP message based service composition is not suitable for dynamic systems that require complex message exchange during composition. Hence, the requirement of an agent-based framework suitable for implementing adaptive complex conversation oriented message exchange has been emphasized. In [217] the Web Service Conversation Language (WSCL) has been proposed to address some of the issues regarding complex messaging. The language is capable of formally representing the order of conversation and the format of input/output. Many of the works have been based on the *AgentCities* platform [218 - 219]. This platform is a large open society of FIPA compliant agents. It consists of 14 backbone platforms across the globe. Currently it hosts 80 active agent frameworks that host services as agents. Most of these frameworks are reported to be

JADE based. The underlying network topology is star-based where the central node is responsible for providing agent platform directory, agent directory, and service directory. A polling agent is responsible for updating the directories whenever a new agent is registered.

In [227, 233] an agent-based service workflow enactment framework has been proposed. The framework utilizes the concept of *social dependence* [228] and *first order ability* [233] to model inter-agent relationship. The framework is built on top of the *AgentCities* platform. The architecture comprises of a set of LEAP-based Mobile Agents [234] that discover the web services in the system and incorporates them as their dynamic behaviors, a centralized Wherehoo server that stores DAML-S service descriptions, a Home Server which is essentially a Piccola (a service composition language [235]) based composition engine and also does DAML-S to Piccola translation. The major disadvantages with this architecture is that: (i) it is dependent on a centralized composition engine and hence, creates a single point of failure, and (ii) the mobile agents have to go through a computationally intensive service discovery process every time they sense a query and iteratively consult the composition engine. Another *AgentCities* based framework is proposed in [220]. It has been identified in this work that the current *AgentCities* framework is not robust and scalable because of the star-based topology. Moreover, all operations needed for a service composition process is totally dependent on the success of the platform directory agent that maintains a global knowledge of the system. Using the specific case study of a conference organizer system the authors proposed a event based agent framework where an Event Organizer Agent analyzes consumer query (represented as task templates) and starts conversing with other service agents accordingly. In [226] an agent-based

architecture is proposed that comprises of a Composite Agent responsible for identifying participant services in a composition, a Service Agent that represents each service instance, and a Master Service Agent that is responsible for tracking service agents and processes a composition plan that has been computed by the Composite Agent. The main limitations of this framework are: (i) the composition process is centralized at the Composite Agent and hence not scalable, and (ii) the Master Agent has to maintain a global system knowledge which is computationally expensive.

The frameworks that have been discovered so far is service-centric since the agents are created to represent the individual services and the composite service that is computed as a product of a service composition process. In contrast there are some frameworks that model the problem more from a user-centric perspective. In [223 - 224] a user-centric agent framework has been proposed based on the ARGUGRID project [225]. The framework leverages argumentative agent technology for inter-agent message exchange during service composition. The architecture comprises of a set of User Agents that represent consumer queries, and a set of Service Agents representing services. Agents are modeled along the BDI (Belief-Desire-Intention) architecture. Each agent has a set of logically represented workflows along with a history of all past decisions and communications stored in its belief. User Agents are responsible for deciding an optimal composition and best service provider. For this an User Agent maintains a global knowledge of service types and service providers in registries. Another user-centric approach can be found in [236]. In this work along with the User Agents and the Service Agents a whole set of mediator agents have been proposed. The architecture includes a Composite Service Agent (CSA) that is responsible for

triggering and runtime monitoring a composite service specification, a Broker Agent (BA) for inter-communication translation issues, Service Matchmaker Agent (SMA) for service matchmaking, and Service Discovery Agent (SDA) for discovery Service Agents on behalf of the User Agents. In [237] a task-ontology based service composition framework has been proposed. The task ontology contains definitions of different pre-defined task-based queries. The architecture consists of Service Requestor Agents (SRA) for representing the queries and Service Provider Agents (SPA) that are responsible for decomposing a complex query based on the task ontology. Each agent also keeps a Fellows' Capability Expectations Matrix (FCEM) that is used to maintain the agent's belief about the expected capability of another agent to provide service to it during its course of action. This helps the agent to eliminate unnecessary conversations or matchmaking during a service composition process. However, the proposed framework suffers from certain weaknesses. First, the task ontology restricts consumer queries that can be decomposable within the ontology space. Secondly, each agent has to keep an FCEM that can grow very large over time. Thirdly, it seems that the FCEM assumes that a SPA is either capable of providing the required service in full or is not capable at all. However, there may be SPAs that partially satisfy the required service and can form a collaboration to provide the complete service. Fourth, since capability is defined as an ordered pair of number of recorded transaction with an SPA and the expected success of making a negotiation with the SPA it is not very clear how during finalizing a deal two capability estimates are compared. Finally, the service composition procedure entails that an agent makes a deal based on its local knowledge of all possible matching SPAs who can accept the deal. If this step fails then the agent asks the Matchmaker Agent for further

results. This may not guarantee optimal composition because the dynamic nature of the system may cause newer and better SPAs to be added in that are not known to the agent. Also the granularity of negotiations is too high for meeting the service latency that the consumer expects. A summarized comparative study of some of the prominent related works with SMARTSPACE has been given below:

Table 8: Comparative Study of SMARTSPACE with other Agent Models

Features	**SMARTSPACE**	**AgentCities based platforms [218 - 220]**	**WSCL based platforms [217]**	**Workflow enactment based platforms [227, 233]**	**ARGUGRID based platforms [223 - 224]**
Service Directory	Decentralized	Centralized	Centralized	Centralized	Centralized
Service Composer	Decentralized & Distributed	Centralized	Centralized	Centralized	Decentralized & Localized
Global System State Maintenance	Not needed	Centralized	Centralized	Centralized	Decentralized & Localized
Service Composition	Runtime	Pre-planned	Pre-planned	Pre-planned	Pre-planned
Service Discovery	Highly Localized	Exhaustive	Exhaustive	Exhaustive	Exhaustive
Query Format Complexity	Simple	Complex (task template)	Very Complex (overkill)	Complex (task template)	Very Complex (overkill)
User-Centric Service Composition (User Agents)	Yes	No	No	No	Yes
Service-Centric Composition (Service Agents)	Yes	Yes	Yes	Yes	No
Optimization Strategies	Included	None	None	None	None
Empirical Analysis	Done	Rare	None	None	None

6.3 Limitations of Centralized Service Composition

In general centralized service composition approaches have some major limitations that may not, in reality, be able to guarantee the best possible composition. In this section we discuss each of them as follows:

- **Single Point of Failure**: A middleware dependent centralized approach is obviously vulnerable to the problem of single point of failure. This is because the if the middleware hosting node in the network crushes or is overloaded with too many user requests then a new user request will be totally lost. This is true even for the *ALNet* middleware where the global *ALNet* instance needs to be stored by the middleware and periodically updated. Also all notifications regarding update are made by the middleware. In the case of the middleware failure it is entirely up to the registered service vectors to interpret the user event and coordinate among themselves so as to avoid redundant discovery of dependency paths (since an abstract node contains several possible service vectors that interpret an event in the same way). This is an extremely complex and computationally expensive process and is currently not incorporated into the current *ALNet* system.
- **Stateless Composition**: Services in general are stateless. This means that the state value produced by a service is not stored by the service. In such a situation the entire runtime composition process has to be synchronous. However, an optimal composition discovered by the middleware does not guarantee such a synchronicity during runtime. This is because runtime environment includes several factors that are not taken into account by the middleware. One such important factor is network load that might lead to restarting the entire runtime composition if a causal service output cannot be received by

a busy responsive service. Another factor is service latency that might cause a responsive service to drop an event-handling process if one of its causal services has a delay to generate output while some other causal services have already generated their output.

- **Conditioned Causality in System**: Service causality within an SOA-based system is conditioned. This means that a service can only respond to its causal services if the pre-conditions required for the service to work (such as availability, computing resource constraints, etc) is satisfied during runtime. However, during discovery of the optimal composition the middleware does not account for all possible internal system events at runtime that might dissatisfy such service pre-conditions. This is mitigated to some extent by the *ALNet* platform since runtime forward traversal over a selected SB only service vectors that have favorable conditions can interpret and respond to a particular causal service event so as to carry the event-handling process forward. However, the middleware always has to supervise such runtime event-handling over a selected optimal SB since it needs to select the next best service vector in situations where the best service vector constraints are unsatisfied. This creates a global lag in the service latency.
- **Dynamic Set of Services**: Any large-scale open SOA-based system is subject to a lot of addition and deletion of services in a very non-deterministic way. Hence, the best composition computed by the middleware may not truly represent the most optimal. New services with better cost and higher user satisfiability may be added after the composition process. Also existing best services may be deleted after the middleware discovers the best composition. Even though the *ALNet* platform can handle the deletion

case with the intervention of the middleware (as described previously) however, it cannot handle the addition case that well. This is because composition process starts off after determining the source and end services. Hence, new and better end services may be overlooked during the event-handling process.

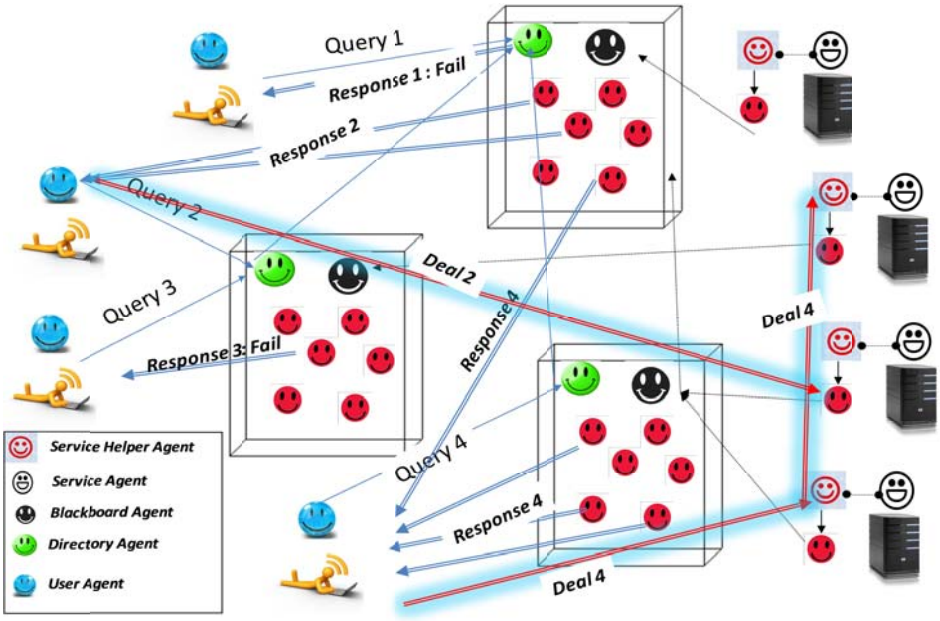


Figure 54: SMARTSPACE - System Overview

6.4 SMARTSPACE: Architectural Overview

The SMARTSPACE network overlay architecture is a hybrid P2P structure that includes a set of dedicated specialized nodes and a dynamic set of peer nodes where services can be hosted by service providers (figure 54). These peer nodes may get added to the system or may drop off from the system non-deterministically. The underlying assumption in the proposed model is that all peer nodes have network access to at least one of the specialized nodes while the specialized nodes must be strongly connected to each other at

any given time point. Each node (specialized and peers) is installed with a JADE container where software agents can be created and maintained. The peer nodes host the *service agents* (*SA*) and the *user agents* (*UA*). The specialized nodes host special agents called in general as *helper agent*s. Specialized nodes are the start-off points for any event-handling process although, unlike centralized middleware systems, they do not participate in any decision making. All kinds of decision making within an event-handling process is done in the peer nodes. If a peer node fails then agents living in that node move to other peer nodes. On the other hand if a specialized node fails then agents living in that node move to a new node. The helper agent that has now moved to the new node then lets other existing specialized node to know about the failure and also the address of the new node. The assumption behind the model is that at no given point of time should all the specialized nodes fail simultaneously. This assumption is important because peer nodes are not required to get registered with the specialized nodes in the *SMARTSPACE* model. Each peer node just needs to know the list of specialized node addresses only. While the architecture can be extended to relax the aforesaid assumption we do not include that in this current work.

Event-handling within the *SMARTSPACE* context is essentially a game problem instance where an *UA* first tries to lookup *SA*s that can satisfy it and then starts a dealing process with each of them. After the dealing process is over the *UA* receives a set of bids from these *SA*s and then confirms the best possible bid. However, this game instance involves several other internal agent behaviors so as to guarantee a near-optimal deal. Before any event-handling can be done the *SMARTSPACE* platform invites service providers to register their servers with *SMARTSPACE*. The registration process includes

installing the *SMARTSPACE* service provider module into their server that hosts their services. Once installation is done SMARTSPACE converts the hosted services into software agents called *service agents (SA)*. *SA*s are executable and lives in JADE containers installed in the host peer (mostly owned by the service provider) during registration. An *SA* inherits all the basic properties of a JADE agent while retaining the service functionality (as its behavior) and its corresponding description (written in OWL-S). The description is reformatted into *g-array* (see chapter 3). An *SA* starts execution only if it wins a deal from a particular *UA*. Once execution is done the output is given back to the *UA* so that it can access it whenever it is ready.

After an *SA* is created SMARTSPACE creates a corresponding *service helper agent* (*SHA*) that attaches itself to the *SA*. The *SHA* is basically a light weight agent that contains only the *g-array* of the *SA* and bids for deals in favor of the *SA*. Thus, *UA*s make deals with the *SHAs*. *SHA* has its own belief system that contains its interpretations for different events (i.e. agent behaviors) and also its knowledge of the system (includes measure functions for various QoS attributes such as service reliability, user utility, user preference, etc). The *SHA* has the capability of verifying the reliability of the *SA* that it works for by periodically checking the accuracy of the *SA*'s output once a deal is won by it with respect to the promised output in the *O-array* of the *SA*. The *SHA* clones itself (called *SHA-clone*) that is then sent off to the nearest specialized node. These clones self-organize themselves into an *O-cluster space* (see chapter 4) of agent clones. Thus, the *SHA-clone*s only have the *O-array* of their corresponding *SA*s. Clone clustering, called *SmartCluster*, is the SMARTSPACE version of organizing services for efficient discovery and query processing. The algorithm

will be discussed in detail in later section. An *SHA* kills itself and all its traces including its clone if the corresponding *SA* gets killed. An *SA* can get killed because of a system crash or because a new modified service needs to be uploaded into the server. This leaves no possibility for a false deal with an *SHA* whose corresponding *SA* is non-existent. The design principle of having a separate *SHA* for every *SA* helps in several ways:

- **Maintaining system fairness**: The design helps to maintain the fairness of the system as a whole and also to optimize overall customer satisfaction. It tries to prevent the *SA* from misrepresenting data to the *UAs* during a deal or to obstruct the working of other competing *SA*s by rogue behavior such as virus implantation, eavesdropping, impersonation, etc.
- **Releasing computational overload**: The design also helps to reduce the computational overload of dealing by decoupling the dealing behavior from the service execution behavior of the *SA*. Thus, *SA*s just have to worry about execution while all complexities related to winning a deal is assigned over the *SHA*. Moreover, for all practical purposes *SA*s are dumb as they do not usually come with a belief system of their own powered by reasoning capability. In other words, service providers do not have to tailor-make their services so as to suit the *SMARTSPACE* platform. Instead, *SMARTSPACE* helps to provide the extra layer of intelligence by creating the *SHA*s for them.
- **Handling confirmed deal break-up**: It may happen that during execution an *SA* gets killed. Under such situation there is no way for the agent that confirmed the deal with the *SA* to know that the deal has to be terminated and a new deal has to be confirmed. *SHA*

provides an efficient solution where in such a situation it can let its deal-making agent know that it can no longer serve it before it gets killed itself.

SMARTSPACE also provides an user module that can be installed into the consumer peer machine. The module provides an interface for users to give their queries in the *DQM* format. The moment a user query is submitted a corresponding *user agent* (*UA*) is created by *SMARTSPACE*. The *UA* is also a lightweight agent that contains the *g-array* of DL-encoded query and maintains its own belief system. The belief system incorporates the user's profile (preference, location, etc) and knowledge of the users utility function variables. The *UA* remains active so long as the user request is not satisfied (if it is satisfiable) or if it finds out that the request is not satisfiable within a stipulated time as required by the user. After a set of *SHA* is discovered the *UA* begins deal with all of them before confirming a deal with the best one of them.

Apart from the *UA*s and the *SA/SHA*s the *SMARTSPACE* platform provides a whole set of helper agents that live in the specialized nodes. There are two kinds of helper agents within the *SMARTSPACE* platform: (i) *blackboard agent* (*BA*), and (ii) *directory agent* (*DA*). Each specialized node contains a *DA-BA* pair. For a given SOA-system we can set up *n* number of specialized nodes that will contain *n DA-BA* pairs. A *BA* helps to form and manage the *O-cluster space* of *SHA-clones* that is formed in its specialized node. For this purpose it keeps its own record of *O-taxonomies* in a special table called the *BA-directory*. The *DA*s, on the other hand, are the first point of contact of the *UA*s and *SHA'*s when they need their desires to get satisfied. For this purpose it uses its own special directory called *DA-directory* that keeps a global summarized overview of all the *O-cluster spaces* in the

entire system. The *DA-directory* is an extremely efficient lookup directory that is based on *DL-Encoding* of *UA/SHA* desires (i.e. the *Q-T1* component) and the *SHA-clone O-array*s. It helps the *DA* to quickly identify whether the given desire can at all be satisfied and if so then which specialized node the query should be redirected and thereby processed. It is to be noted here that during an event-handling process a *SHA* can also start deals with other *SHA*s if it does not get its own desire (i.e. *Input*) satisfied by the *UA* that starts off the event-handling process. Hence, in general any agent that needs to get its desire satisfied to achieve its goal has to contact one of the *DA*s. *DAs* redirect the desire to the *BA* of the specialized node where the desire can be satisfied. The job of the *BA* is to map the desire (i.e. Q-T1 query) into the O-cluster space of *SHA* clones that it maintains. This is basically the service discovery phase and the corresponding algorithm, called *SHASniffer*, is going to be discussed in later section.

Each agent and its roles will be discussed in greater depth in the following sections. We will also provide the algorithm that the agents follow in executing each role along with formal analysis.

6.5 Directory Agent

The *directory agent* (*DA*) is a very important agent within the *SMARTSPACE* framework since it helps to improve the efficiency of event-handing significantly without agents to flood their queries for service discovery or multi-cast queries to the specialized nodes for service discovery over a large space of *SHA-clones*. The *DA* maintains a specialized directory, called the *d-directory*, that helps it to: (i) understand the satisfiability

of an agent desire and (ii) find out the correct *BA* that can process the desire. Both the operations are done by the *DA* in worst case constant time. We first discuss the *DA-directory* data structure in the following sub section.

6.5.1 DA-directory

The *DA-directory* is table of N 2-order tuples $\langle DA\_ID, MC_{DA\_ID}\rangle$ where N is the total number of *DA*s in the system, *DA_ID* is the unique ID of a *DA*, and MC_{DA_ID} is the *Mash*

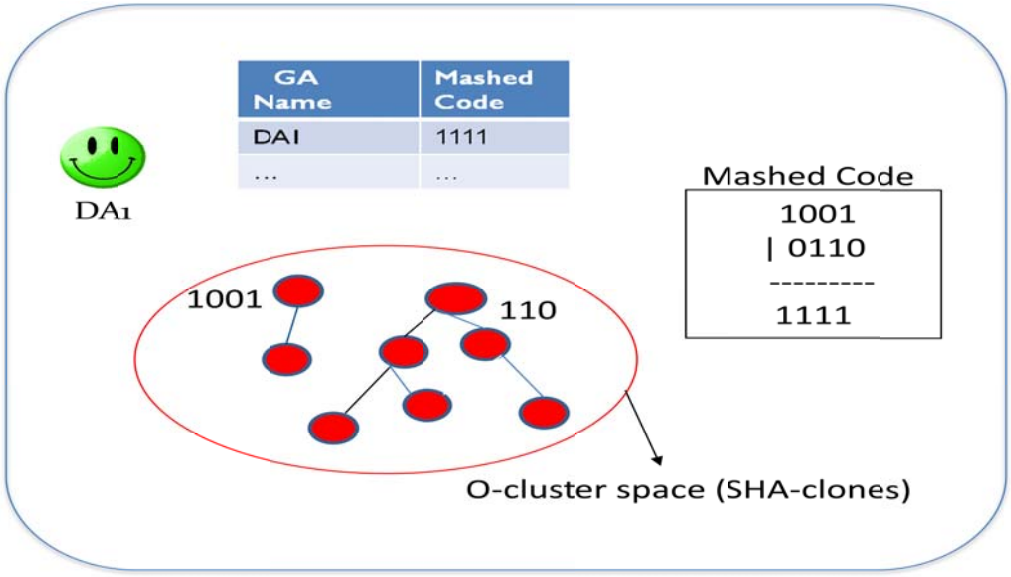


Figure 55: DA-directory as Dynamically Maintained by a DA

Code of the O-cluster space of *SHA-clones* that reside in the container of the corresponding *DA* (figure 55). Thus, if there are 4 *DA*s in total within a system then each *DA* will have 4 rows in its *DA-directory* corresponding to each of the 4 *DA*s. The *Mash Code* (*MC*) associated with a particular *DA* is *n*-ary OR of the DL-codes of the root *SHA-clone*s of the *O-cluster space* within the specialized node where the *DA* lives. The root *SHA-clone* represents a particular *O-taxonomy* within this *O-cluster space*. Thus, the *Mash Code*

contains all the significant 1-bits of each root *SHA-clone* and hence, represents the entire *O-cluster space*.

It is to be noted that according to the definition of a *DA-directory* each *DA* holds an identical copy for itself. Hence, if there is any update within a particular row of the *DA-directory* then all the other *DA*s need to be notified about the update. Update in *DA-directory* can take place for three reasons: (i) *MC* changes because of a new *O-taxonomy* addition into the corresponding *O-cluster space*, (ii) *MC* changes because of an existing *O-taxonomy* deletion within the corresponding *O-cluster space*, and (iii) *DA* gets killed because of a node crash. The second case is a relatively rare event since the *O-cluster space* represents the functional category of a service and categories do not disappear so often. The third case is also not a very frequent event for specialized nodes and requires only deletion of corresponding tuple in the *DA-directories* of other existing *DA*s. It is important to note that the *DA-directory* is not affected by the large number of dynamic addition and deletion of *SHA-clones* in the *O-clusters* since the *DA* is only interested in the taxonomy information. Thus, *DA_directory* update is not an expensive process.

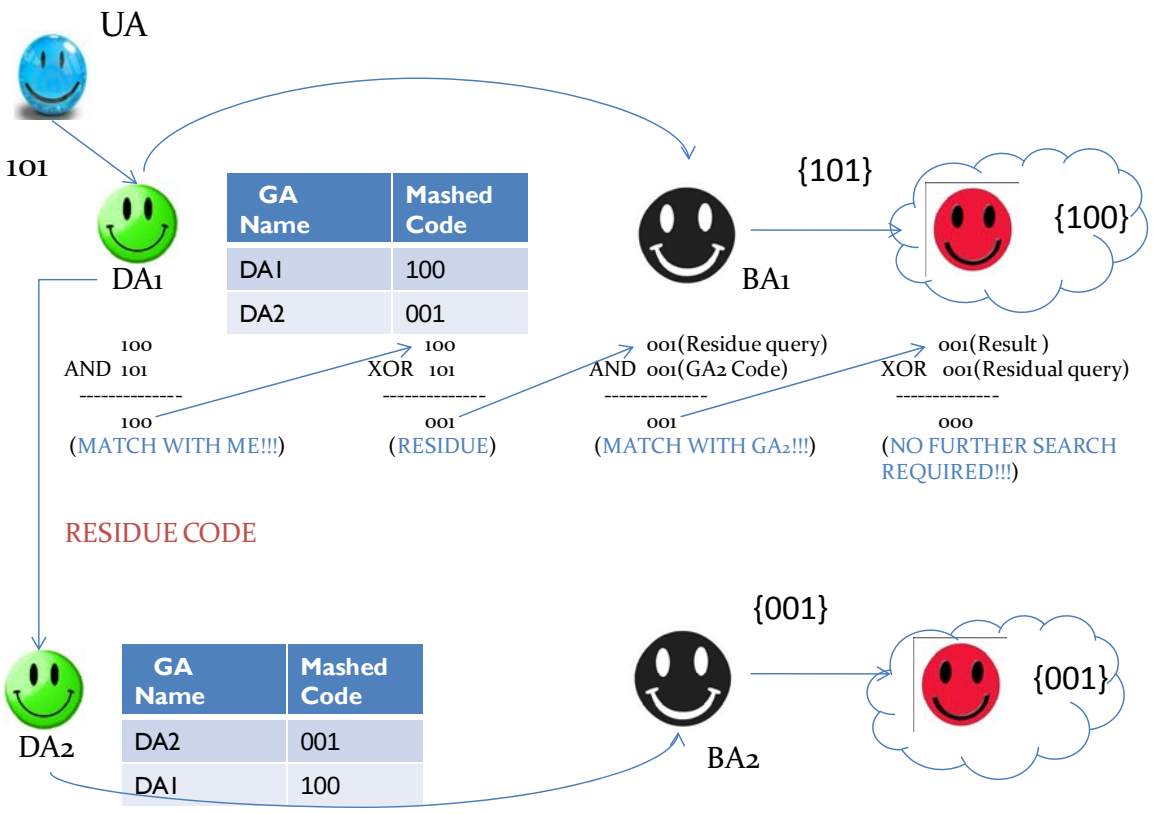


Figure 56: *SmartDirect* Process Overview

6.5.2 SmartDirect: Query Mapping Algorithm

Using the *DA-directory* the *DA* efficiently recognizes the satisfiability of a desire (formally a Q-T1 query; see chapter 3) of an agent. It is to be understood at this point that since all *DA*s have the same identical *DA-directory* hence, an agent just needs to request only one *DA* with a desire. When an agent requests a *DA* the *DA* first tries to check up whether the *SHA-clones* available in its own container can fully satisfy the Q-T1 query. To do that it first does an *AND* operation over its own *MC* and the DL-code of the Q-T1 query (figure 56). There can be two situations under such circumstances: (i) failure, (ii) partial success and (iii) complete success.

- **Failure**: If the operation results in 0 then there is no match at all. In this case the *DA* looks into the remaining tuples to find a match. If all the tuples end up giving a 0 as the *AND* result then the *DA* lets the agent know that there is no *SHA* (and hence, no *SA*) that can satisfy its desire at the moment. However, the *DA* can still keep the desire in its memory if the agent is willing to spend some more time in the hope that some *SHA* might possibly join a cluster space and form a new taxonomy altogether.
- **Partial Success**: If the *AND* operation does not result in a 0 then the *DA* does an XOR operation with the result and the DL-code of the desire (figure 56). If the XOR result is not 0 then there exists a partial match between the desire of the requesting agent and the available *SHA-clones* in the *DA*'s container. In such a case the *DA* then takes the XOR result and matches that with the other *MC*s in its directory. The XOR result is essentially the left-over 1-bits of the desire that are still to be matched.
- **Complete Success**: If the XOR result in the previous case is 0 then the *DA* knows that the *SHA-clones* inside its own container are able to satisfy the desire completely. In the case of a complete success the *DA* does not need to look into the other tuples for match because of the exclusive existence of a particular *O-cluster space* in a single specialized node. More about this property will be discussed in later section.

It is be noted, however, that in any kind of success (partial and complete) a match may imply that there may exist a *strong solution set* (*SSS*) or a *weak solution set* (*WSS*) to the desire. Once the *DA* knows about the satisfiability of a desire and also the container where the desire, if satisfiable, has a solution set it redirects the query to the *BA* agent for further processing of the desire. The underlying algorithm that the *DA* follows to map a

query is called *SmartDirect*. We now prove that the *AND* followed by the *XOR* operation is a sound and complete method to understand the satisfiability of a desire.

Theorem 6.1: Given a DL-encoded Q-T1 Q and a set of *MC* (say S_{MC}), Q is satisfiable iff $\exists M \in S_{MC}, (Q \wedge M) \neq 0$.

Proof: Since M contains all the significant 1-bits of the root *SHA-clones* hence, if there is a exists a corresponding 1-bit of M in Q then *AND* operation produces a corresponding 1-bit. Hence, there exists a *O-taxonomy* corresponding to the significant 1-bit of M that has *g-relation* with Q. Therefore Q is satisfiable.

If we assume that Q is satisfiable then Q must have a *g-relation* with at least one M in S_{MC}. *g-relation* implies that there must exist a match that is either: (i) exact, or (ii) *plug-in*, or (iii) *subsume,* or (iv) *sibling*. According to definitions, for any of these matches there must exist at least one-pair of corresponding 1-bits. Hence, $\exists M \in S_{MC}, (Q \wedge M) \neq 0$.■

We now prove that *XOR* is a sound and complete test for partial success as follows:

Theorem 6.2: Given a DL-encoded Q-T1 Q and a set of *MC* (say S_{MC}), there exists a partial success with a *DA* (say D_X) iff $\exists \langle D_x, M \rangle \ni (Q \wedge M) = R \neq 0 \rightarrow R \vee Q \neq 0$.

Proof: If $\exists \langle D_x, M \rangle \ni (Q \wedge M) = R \neq 0 \rightarrow R \vee Q \neq 0$ is assumed to be true then: (a) R must have at least one 1-bit whose corresponding bit in Q is a 0-bit or (b) Q must have at least one 1-bit whose corresponding bit in R is a 0-bit. The first case is contradictory since R is an *AND* product and must contain all the matching 1-bits of Q with M. Therefore, the second case must be true. If that is so then there are 1-bits in Q that are yet to be matched. Therefore, Q has a partial success with D_X.

If we assume that Q has a partial success with D_X then Q must have at least one 1-bit whose corresponding bit in M is a 0-bit. Thus, R will not contain this 1-bit of Q. Hence the *XOR* operation will output at least one 1-bit corresponding to the unmatched 1-bit of Q. In other words, $\exists\langle D_x, M\rangle \ni (Q \wedge M) = R \neq 0 \rightarrow R \vee Q \neq 0$ is true. ■

Based on these two theorems the algorithm is outlined in figure 57.

```
==============================
ALGORITHM: SmartDirect (DA Behavior)
Input: Q-T1 [DL-code]
Output: Success/Failure [Boolean]
==============================
START
Message receipt = readMessage()
IF receipt.sender == UA or SHA && receipt.content == Q-T1
        FOR each row in DA-directory
                IF hasMCMatch(Q-T1, MC[i]) == complete success
                        sendMessage(BA[i], Q-T1)
                        Failure_flag = 0
                        return Success
                END IF
                ELSE IF hasMCMatch(Q-T1, MC[i]) == partial success
                        sendMessage(BA[i], Q-T1)
                        Failure_flag = 0
                        continue
                END IF
                ELSE IF hasMCMatch(Q-T1, MC[i]) == failure
                        Failure_flag = 1
                        continue
                END IF
        END LOOP
END IF
IF Failure_flag = 1
        Return Failure
END IF
END
```

Figure 57: SmartDirect Algorithm

6.6 Blackboard Agent

The *blackboard agent* (*BA*) is the second kind of helper agent within the *SMARTSPACE* framework that acts in two different ways: (i) to learn service categories and

organize *SHA-clones* into an *O-cluster space* and (ii) to map an agent query (redirected from a *DA*) into its *O-cluster space*. Since the *O-cluster space* is highly volatile with new *SHA-clones* joining it and old ones leaving the *BA* maintains it using its own directory called the *BA-directory*. The BA-directory maintenance job involves several objectives:

- One objective of maintenance is to improve the clustering of the *SHA-clones* and also query mapping by decreasing the number of required communications.
- The second objective is to let the *DA*s know if a new taxonomy has been added or an old one has been deleted.
- The third objective is to make sure that a specialized node is not over crowded with too many *SHA-clones*. In other words, the *BA* makes sure using the *BA-directory* that there is a equitable distribution of *SHA-clones* over all the specialized nodes thereby improving query mapping performance and reducing node overloading.

In order to understand how these objectives are met we first provide a detailed discussion of the *BA-directory* itself.

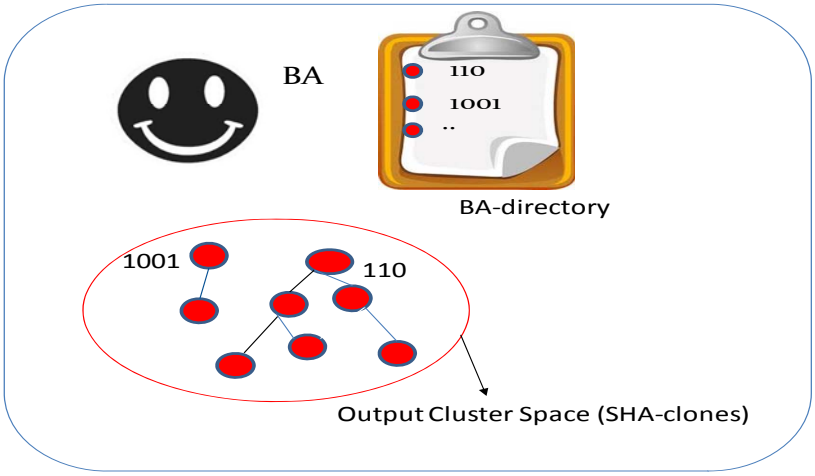


Figure 58: BA-directory as Dynamically Maintained by the BA

6.6.1 BA-directory

The *BA-directory* is table of N 2-order tuples $\langle SHA\text{-}clone\_ID, DL\text{-}code(SHA\text{-}clone\_ID)\rangle$ where N is the total number of *O-taxonomies* in specialized node where *BA* lives , *SHA-cone_ID* is the unique ID of a root *SHA-clone*, and *DL-code(SHA-clone_ID)* is the corresponding *DL-code* of the *O-array* of the *SHA-clone* (figure 58). Unlike the *DA-directory*, the *BA-directory* is completely unique to a particular *BA* and is not shared or copied. Hence, there is no requirement of global system updates when new root *SHA-clones* join or old ones leave. In other words the *BA* is a very localized agent having a local specific perspective while the *DA* has a global yet abstract perspective. Although the *BA-directory* is a localized directory yet it is significantly important to note that the directory does not contain the information of the entire *O-cluster space*. In other words, unlike centralized models (including the proposed *ALNet* framework), the services (i.e. the *SA*s) do not have to get registered with the *BA*.

6.6.2 BA-directory Update

The *BA-directory* undergoes a lot more update than a *DA-directory*. This is because an open system dynamics entails higher probability of joining and leaving of root *SHA-clones* as compared to creation and extinction of taxonomies as a whole.

Increment Update: Whenever a new root *SHA-clone* is identified by the *BA* it has two options to take: (i) to create a new tuple and insert the root information, or (ii) to modify an existing tuple and update the root information. The first option is taken if:

- There exists a root entry in the directory such that the new root *SHA-clone* has a *sibling* match with it and such that it does not have non-sibling match with any root entry.

or

- For all root entries in the directory the new root *SHA-clone* has no *g-relation*. It is in this case only when the *BA* reports to the *DAs* that a new taxonomy has been formed.

The second option is taken if neither of the given conditions holds. In other words, the new root *SHA-clone* has a *subsume* match with at least one root entry. In such a case the matched root entry is updated with the new root information. However, in this case the *DAs* do not need to be reported since no new taxonomy is formed.

Decrement Update: Whenever an old root *SHA-clone* leaves the cluster space (mostly because it has to die) then the BA just has to remove its corresponding tuple from the *BA-directory* and insert new tuples that correspond to the old root's immediate children within its taxonomy. The *BA* only reports to the *DA*s if there is no child *SHA-clone* i.e. the taxonomy itself gets deleted.

In the next section we discuss the *SHA-clone* clustering algorithm, called *SmartCluster*, and how *BA* helps to initiate and improve the clustering process.

6.7 SmartCluster: Distributed STC Algorithm

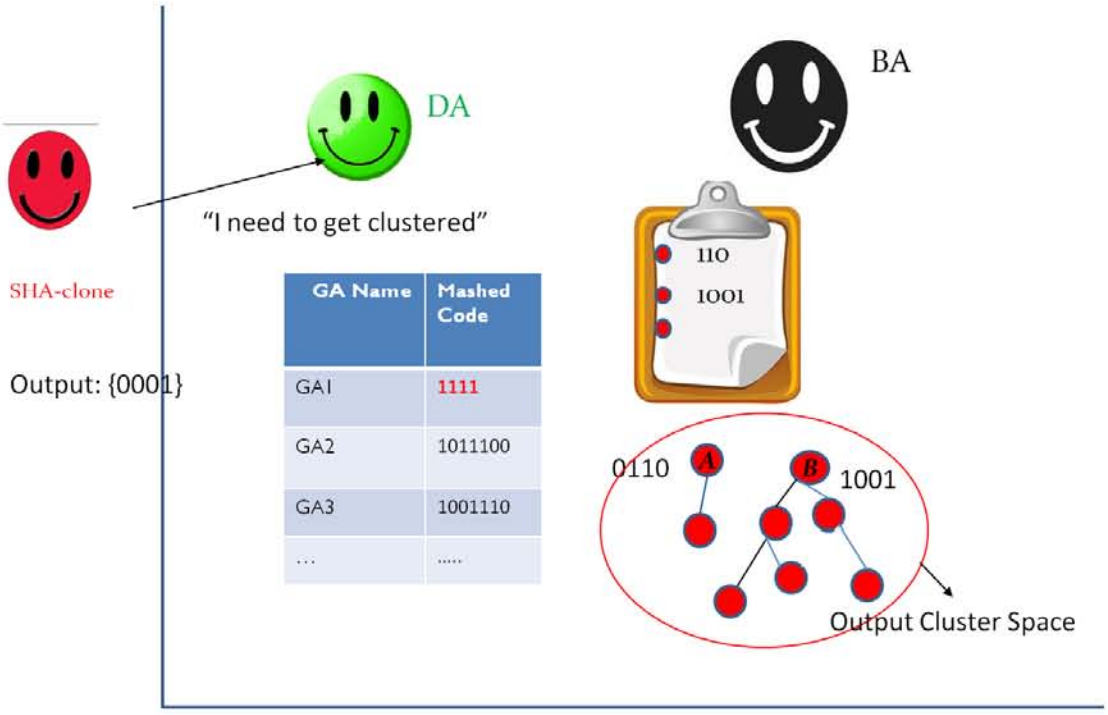
SmartCluster is a distributed version of the *Semantic Taxonomical Clustering* (*STC*) elaborated in chapter 4. There are some major differences between the two as follows:

1. *STC* is a clustering of service vectors while *SmartCluster* is a cluster of the active *SHA-clone* agents.
2. *STC* is processed by the *ALNet* middleware. On the other hand *SmartCluster* is a completely distributed process based on mutual communication between *SHA-clones* and the helper agents (*DA* and *BA*).
3. *STC* is a process that forms both the *O-cluster space* and the *I-cluster space*. In contrast to that *SmartCluster* only forms the *O-cluster space*. The *I-cluster space* is not needed within the *SMARTSPACE* framework although it is a integral part of the *ALNet* framework. Whenever an *SHA* creates its clone (i.e. the *SHA-clone*) and sends it to the nearest available specialized node the *SHA-clone* communicates with the *DA* living in the node to know whether it can live in the existing *O-space*. The *DA* checks whether there exists any existing taxonomy that can include the new *SHA-clone*. This checking is just the same *AND-XOR* operation dual that the *DA* performs for a query redirection. There are three possibilities after this operation is done: (i) the *SHA-clone* has no *MC* match at all, (ii) the *SHA-clone* has a single *MC* match, and (iii) the *SHA-clone* has multiple *MC* matches. We describe each of the cases as follows:

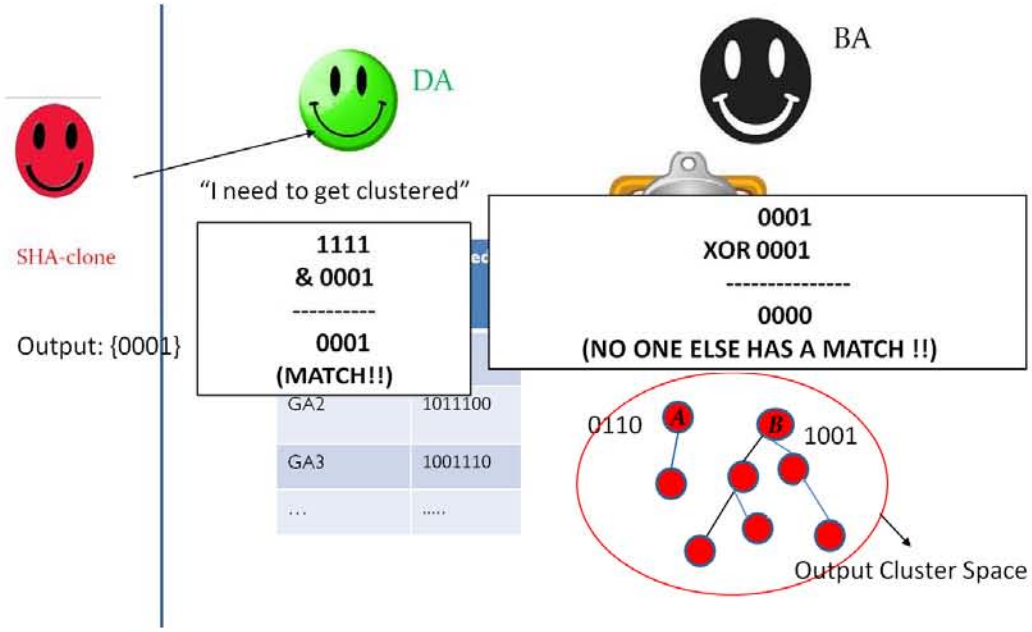
- **No *MC* Match**: This case implies the *SHA-clone* has to form a new taxonomy of its own. In this case it first asks the *BA* whether there is enough room (memory space) in the current specialized node. If yes the *BA* inserts a new tuple in its *BA-directory* and the *SHA-clone* starts living in the specialized node. The *DA*s update their *DA-directories* accordingly. If there is no room then the *SHA-clone* moves to the next available specialized node and starts communicating in the same way with the *DA* living there. It is to be noted here that the *SHA-clone* has to start the same protocol all over again since it might happen that during its move to the next specialized node a similar kind of *SHA-clone* has already started living in one of the specialized nodes. So the *SHA-clone* has to ensure that it does not start a redundant taxonomy (i.e. similar taxonomy w.r.t *g-relation*) in multiple specialized nodes. This property of *SmartCluster*, as mentioned earlier in section 6.5.2, is called *exclusive residence of O-cluster space*. This property is very important since redundancy will increase the search space for solution set when a query is mapped. Also the *DA*'s performance for identifying a query's satisfiability gets negatively affected since it cannot ensure complete 'exclusive' success even if the *XOR* product is 0.
- **Single *MC* Match**: This case implies that the new *SHA-clone* has to stay only in the current specialized node. In other words, it has an exclusive *g-relation* with one or more taxonomies present in the current *O-cluster space*. This happens only when there is a complete success in the *DA-directory*. In such a situation the *DA* communicates with the corresponding *BA* and asks it to search for all those matching taxonomies. The *BA* then looks up its *BA-directory* tuples one by one and does a *g-*

subsumption match. In this way, as explained in section 6.6.2, the *BA* filters out all root *SHA-clones* that have *g-relation*s with the new *SHA-clone*. After the filtering process the *BA* then notifies all these *SHA-clones* about the new *SHA-clone*. Each individual root *SHA-clone* then tries to understand the kind of *g-relation* it has with the new arrival. If the *g-relation* is *subsume* then it just tells the new *SHA-clone* to consider it as its new child. If the *g-relation* is *plug-in* then it sees whether the new *SHA-clone* can be a parent or a sibling of its current children *SHA-clones*. If the test (called *test of parenthood*) is positive then it tells the new *SHA-clone* that it is now its new parent and also tells the affected children *SHA-clones* that they have a new parent (i.e. the new *SHA-clone*). However, if the test is negative then it just tells its immediate children *SHA-clone* to repeat the entire test of parenthood individually. Thus, from an individual point of view a particular *SHA-clone* only communicates with its immediate children and the new *SHA-clone*. Thus, *SmartCluster* is a truly self-organizing form of *STC*.

- **Multiple *MC* Match**: This case implies that there are multiple taxonomies existing in separate specialized nodes where the new *SHA-clone* has *g-relation*. In such a situation the *DA* tells the *SHA-clone* to reproduce new clones and send them to those matching specialized nodes. After this operation is done each of the new clones along with their origin *SHA-clone* starts communicating with the *BAs* in the same way as has been explained in the previous match case. The *SmartCluster* algorithm has been illustrated in figures 59 and 60.

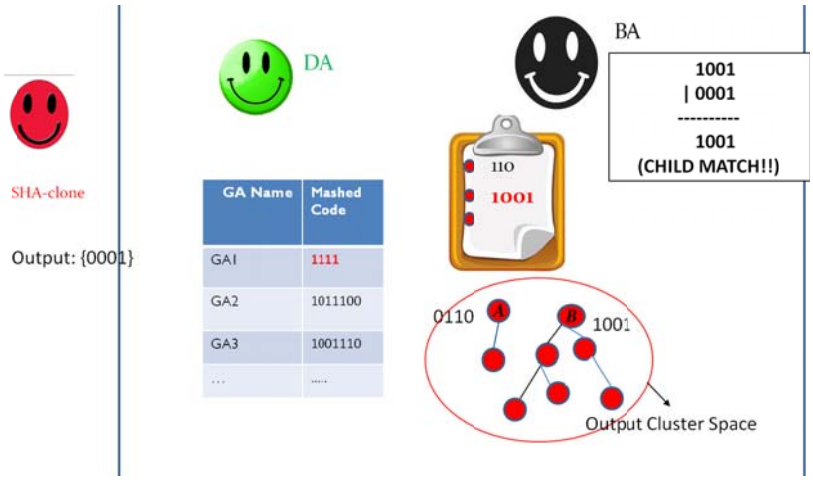


(a)

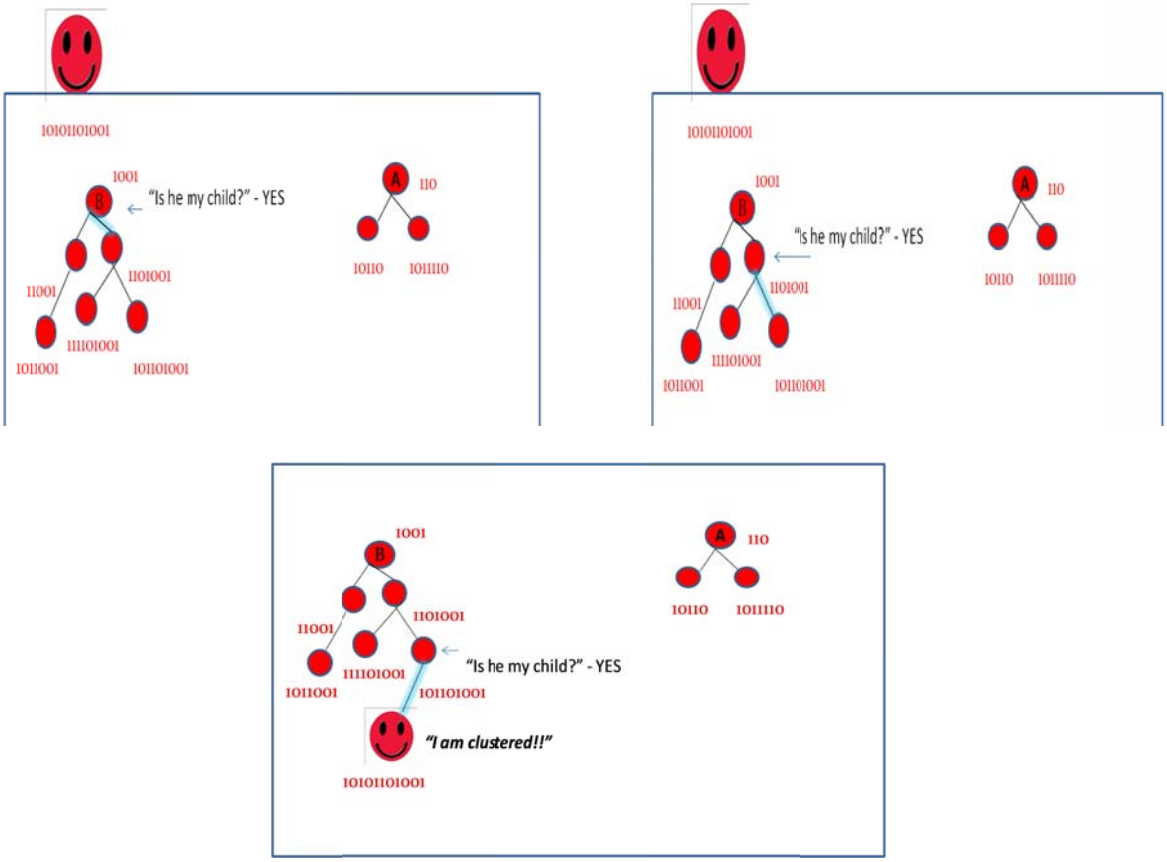


(b)

Figure 59: *SmartCluster* Initiation Process: DA Finds the Correct BA



(a)



(b)

Figure 60: *SmartCluster*: (a) Pruning Search Space, (b) Phases

6.8 SmartMap: Distributed Desire Processing Algorithm

SmartMap is a distributed algorithm for processing a desire (i.e. a *Q-T1* query) when it gets redirected from a *DA* to a set of *BA*s. The algorithm follows the same logic as that of *SmartCluster* except that the goal now is to find a solution set of *SHA-clones* who can satisfy (strong or weak) the desire of the querying agent. The *SmartMap* process is initiated by a *UA*. The *UA* contacts a *DA* and lets it know about its desire. The *DA* then redirects the desire to the set of *BA*s when it finds the desire to be satisfiable. Each *BA* then looks up the *BA-directory* and maps the desire onto the matching *O-taxonomies* of *SHA-clones*. Whenever a *SHA-clone* finds out the type of *g-relation* it has with the desire it immediately contacts the *UA* telling it that its corresponding *SHA* is a bidder and also telling whether the bidder is a strong bidder or a weak bidder. The corresponding *SHA* is a strong bidder if the contacting *SHA-clone* has an *exact* or *plugin g-relation* with the desire. Otherwise the *SHA* becomes a weak bidder. The algorithm for *SmartMap* is given in figure 61.

Note that for an event-handling process even though the *SmartMap* process is initiated by a *UA* yet it can be started up by *SHA*s as well. This is because a bidder *SHA* may find out that it needs to deal with other *SHAs* in the system so as to win the deal. Thus, it starts up a *Q-T1* query by calling up its nearest available *DA*. In the context of *SMARTSPACE* the event-handling process is called *SmartDeal*. We describe *SmartDeal* in detail in the next section.

```
==================================
ALGORITHM: SmartMap (BA Behavior)
Input: Q-T1 [DL-code]
Output: done [Boolean]
==================================
START
Message receipt = readMessage()
IF receipt.sender == DA && receipt.content == Q-T1
        FOR each row i in BA-directory
                IF g-subsumption(Q-T1, RootSHA[i])) == exact/plugin/subsume/sibling
                        Match = g-subsumption(Q-T1, RootSHA[i]))
                        sendMessage(RootSHA[i], Q-T1, Match)
                        continue
                END IF
                ELSE continue
        END LOOP
return done
END IF
END

==================================
ALGORITHM: SmartMap (SHA-Clone Behavior)
Input: Q-T1 [DL-code]
Output: done [Boolean]
==================================
START
Message receipt = readMessage()
IF receipt.sender == BA or SHA-clone && receipt.content.Query == Q-T1
        Match = receipt.content.Match
        IF Match == exact
                sendMessage(Q-T1.sender, strong solution, 1)
        END IF
        ELSE IF Match == plug-in
                sendMessage(Q-T1.sender, strong solution, 2)
        END IF
        ELSE IF Match == subsume
                For each SHA-clone child i
                        IF g-subsumption(Q-T1, i) == exact/plugin/subsume/sibling
                                Match = g-subsumption(Q-T1, RootSHA[i]))
                                sendMessage(i, Q-T1, Match)
                        END IF
                END LOOP
                sendMessage(Q-T1.sender, weak solution, 3)
        END IF
        ELSE IF Match == sibling
                For each SHA-clone child i
                        IF g-subsumption(Q-T1, i) == exact/plugin/subsume/sibling
                                Match = g-subsumption(Q-T1, RootSHA[i]))
                                sendMessage(i, Q-T1, Match)
                        END IF
                END LOOP
                sendMessage(Q-T1.sender, weak solution, 4)
        END IF
return done
END IF
END
```

Figure 61: SmartMap Algorithm

6.9 SmartDeal: Distributed Event-Handling Algorithm

SmartDeal is a completely distributed and asynchronous algorithm for computing the near-optimal composition for a given user event. The process, as mentioned in the previous

section, is initiated by a *UA*. The *UA* keeps a directory of its own, called *deal-directory*, to execute *SmartDeal*. A *deal-directory* is used for several purposes:

- To maintain a record of all bidder *SHAs*. This record is called the *deal list*.
- To maintain a record of all possible candidate *SHA*s. This record is called the *candidate list*. Candidate *SHAs* are those bidder *SHAs* that has accepted the deal or are known a priori by the *UA* to be able to get the deal done.
- To finally confirm a deal by choosing the best possible candidate bidder *SHA* out of the *candidate list*.

It is to be noted that the *deal-directory* is also owned by every *SHA* as well. The reason behind this will be clear when the *SmartDeal* algorithm will be explained in later section. We now first explain the *deal-directory* structure in the next sub-section.

6.9.1 Deal-directory

The *deal-directory* is a table of *N* 3-ordered tuple $\langle QSP, SHA_{bidder}\_Info, MS\rangle$ where *N* is the number of bidder *SHAs* that has to be condidered, SHA_{bidder}_Info is a tuple $\langle SHA_{bidder}\_ID, SHA_{bidder}\_cost\rangle$ where SHA_{bidder}_ID is the unique agent ID of a particular bidder *SHA* and SHA_{bidder}_cost is the cost of the service (i.e. *SA*) that it represents, *MS* refers to the *DQM* match strength (i.e. whether the bidder *SHA* is a strong bidder or a weak bidder), and *QSP* is a special data structure called the *Query-Service Pair*. The *QSP* is defined as 2-ary ordered pair $\langle ID, DL\text{-}code(UA_{desire})\rangle$ where *ID* refers to the unique agent ID of the deal maker agent and *DL-code*(UA_{desire}) is the *DL-code* of the initial *UA* desire. To summarize the *deal-directory* of a particular deal maker agent (*UA* or *SHA*) contains

information about the all the user events that it is currently handling along with the corresponding set of bidder *SHAs* that it requires to deal with for completing the handling process.

The *deal-directory* is partitioned into separate sub-tables for each concurrent user-events that the deal maker tries to handle. For an initiating *UA* there will be only one *deal-directory* table since it specifically handles only one user event. However, for *SHA*s that might have to deal with other *SHAs* there can be several concurrent user event-handling processes that it has got itself involved with. Each sub-table is then categorized according to the 9 possible *MS* values (see chapter 3).

6.9.2 SmartDeal Algorithm

SmartDeal includes three major sub processes in a temporal order: (i) *Make Deal*, (ii) *Accept Deal*, and (iii) *Confirm Deal*. In this section we explain each of them in their corresponding sub-sections.

6.9.2.1 Make Deal

Make Deal is a process that is started off by a deal maker agent immediately after it starts getting responses from *SHA-clones*. To begin with, the *UA* first gets a set of such responses. It then checks up the type of *g-relation* matches that the *SHA-clones* report to it. After that it computes the *g-relation* of its Q-T2 component with the corresponding bidder *SHAs' I-array*. During *g-subsumption* computation over the *I-array* the *UA* notes all the input parameters of the bidder *SHA* that are unmatched. It has to options at this point: (a) to proactively ask the user for that information or (b) to let the *SHA* know that some of the

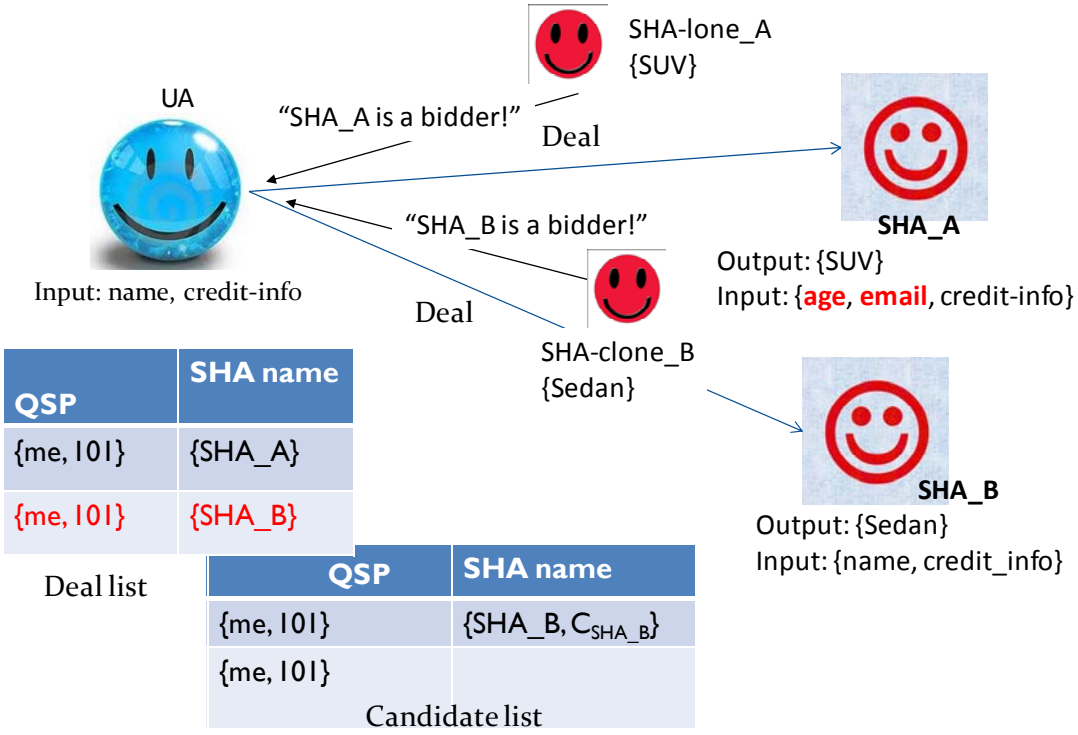


Figure 62: SmartDeal initiation - Make Deal process

the *SHA's* input parameters are unmatched. In this work we choose the second option assuming that in order to keep user-friendliness the first option should only be taken if there can be no event-handling done by the second option.

At this point, as per the *DQM* model, there can be three cases: (i) strong match (*exact* or *subsume*), (ii) weak match (*plug-in*), and (iii) *sibling*. Since the *g-relation* reported by the *SHA-clones* also has 3 possibilities hence, the *UA* computes the appropriate *MS* value out of the 9 possible values (i.e. *SS, SW, SSiS, WS, WW, WSiS, SiS, SiW, SiSi*; see chapter 3). Based on the MS value the *UA* creates a *deal-directory* tuple and inserts the tuple in the correct category. If the *MS* value is *SS* then it keeps the tuple in the *candidate list* since it knows for sure that the bidder *SHA* can definitely accept the deal and execute the deal. All other cases

go into the *deal list*. It is to be noted at this point that the *candidate list* is a special category within a *deal-directory* table where the *MS* value is *SS*.

After a tuple is inserted into a *deal list* the *UA* then starts communicating with the corresponding *SHA* of the tuple. This is called a *deal*. When the *SHA* receives a deal it first creates a *QSP* corresponding to the deal. After that it checks whether any of its input parameters are still unmatched. If the bidder *SHA* has unmatched parameters it then forms up a desire by OR-ing over the unmatched parameters and then sends that desire to the nearest available *DA*. The whole process of *SmartDirect* and *SmartMap* starts off and a fresh set of *SHA-clones* respond to the bidder *SHA* (which is currently the deal maker *SHA*) with new bidder *SHAs*. The deal maker *SHA* then repeats the same deal making process as was done by the *UA* by trying to match the input of the new bidder *SHA* with the given input of the *UA*. This can lead up to a temporally ordered set of deal making which eventually terminates in a bidder *SHA* that does not have to make any further deal (since all its input parameters are matched). Figure 62 outlines an overview of the Make Deal process.

6.9.2.2 Accept Deal

Once a bidder *SHA* does not have to deal any more it replies back to its deal maker and lets it know about its own cost (i.e. SHA_{bidder}_cost). This process is called *accept deal*. When a deal maker *SHA* receives an accept deal from a bidder *SHA* it takes off the bidder *SHA* tuple from the *deal list* and puts it into the *candidate list*. If the *candidate list* is full (i.e. the *deal list* for the user event in question is empty) then it chooses the best candidate bidder *SHA's* cost and adds it up with its own cost. After that it reports the cumulative cost

to its deal maker (which again can be another *SHA*). In this way accept deal process finally terminates with the initiating *UA* receiving the *accept deal* reply (figure 63). After that the *UA* fills up its *candidate list* in a similar way and then chooses the best candidate bidder *SHA*. This process is called *confirm deal* and is elaborated in the next section.

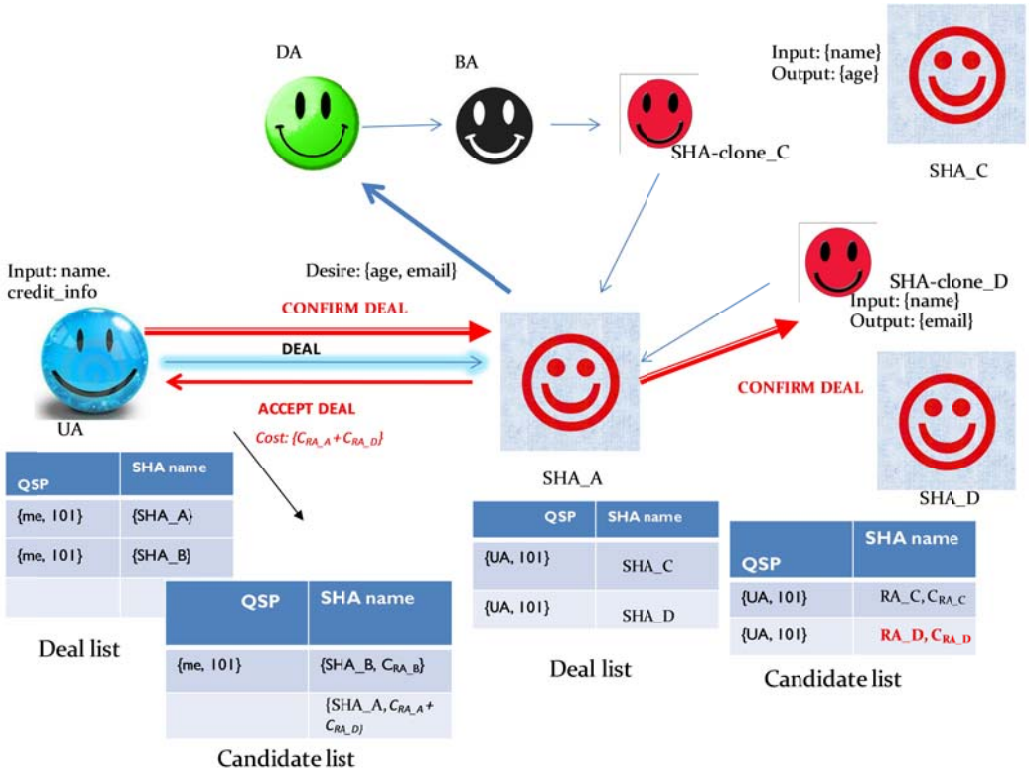


Figure 63: SmartDeal completion - Confirm Deal process

6.9.2.3 Confirm Deal

The confirm deal process, in contrast to the accept deal process, is essentially a forward chaining process starting off with the *UA*. Once a *UA* starts it off by sending a confirm deal message to the best bidder *SHA* the bidder *SHA* then re-checks its *candidate list* to see if any new bidder *SHA* has accepted its deal (if any) and is better than the one it chose while

sending the accept deal message. Therefore re-checking helps *SMARTSPACE* framework to produce more accurate on-the-fly compositions as compared to other middleware oriented centralized system including the proposed *ALNet* framework.

Once the *SHA* chooses the best bidder *SHA* from the *candidate list* it then sends a *confirm deal* message to the corresponding *SHA*. It then waits for this bidder *SHA* to reply back with an output. This process goes on until it reaches the bidder *SHA* that has not made any deal. Once it hits the final bidder *SHA* this final bidder then calls up its corresponding *SA* and tells it to execute by providing it with the input data. After an output is generated the *SHA* then passes the output back to its caller *SHA*. The process again goes backward till it hits the *UA* which finally provides the output of its chosen best bidder *SHA* to the user. The process has been shown in figure 63. In this way the user event-handling process is completed. The underlying *SmartDeal* algorithm is given in figure 64 (a, b, c, d).

```
==========================================
ALGORITHM: SmartDeal (Make Deal Behavior – UA/SHA)
Input: Q-T1 [DL-code]
Output: done [Boolean]
==========================================
START
Message receipt = readMessage()
IF receipt.sender == SHA-clone
        IF receipt.content.match_type = 1 or 2
                /* retrieves unmatched input params */
                IF getInputMisMatch(SHA.input, Q-T2) == NULL
                        Bidder_preference = receipt.content.match_type + 1
                        insertCandidateList(SHA, Q-T1, Bidder_preference)
                END IF
                ELSE
                        Desire = getInputMisMatch(SHA.input, Q-T2)
                        IF getMatch(SHA.input, Q-T2) == subsume
                                Bidder_preference = receipt.content.match_type + 2
                        END IF
                        ELSE IF getMatch(SHA.input, Q-T2) == plugin
                                Bidder_preference = receipt.content.match_type + 3
                        END IF
                        ELSE IF getMatch(SHA.input, Q-T2) == sibling
                                Bidder_preference = receipt.content.match_type + 4
                        END IF
                        insertDealList(SHA, Q-T1, Bidder_preference)
                        /* deals with SHA informing it about its unmatched desire */
                        sendMessage(SHA, MakeDeal, Desire, Q-T2)
                END IF
        END IF
        ELSE IF receipt.content.match_type = 3
                        IF getInputMisMatch(SHA.input, Q-T2) == NULL
                                Bidder_preference = receipt.content.match_type + 4
                                sendMessage(SHA, MakeDeal, NULL, Q-T2)
                                insertDealList(SHA, Q-T1, Bidder_preference)
                        END IF
                        ELSE
                                Desire = getInputMisMatch(SHA.input, Q-T2)
                                IF getMatch(SHA.input, Q-T2) == subsume
                                        Bidder_preference = receipt.content.match_type + 5
                                END IF
                                ELSE IF getMatch(SHA.input, Q-T2) == plugin
                                        Bidder_preference = receipt.content.match_type + 6
                                END IF
                                ELSE IF getMatch(SHA.input, Q-T2) == sibling
                                        Bidder_preference = receipt.content.match_type + 7
                                END IF
                                insertDealList(SHA, Q-T1, Bidder_preference)
                                sendMessage(SHA, MakeDeal, Desire, Q-T2)
                        END IF
                END IF
                ELSE IF receipt.content.match_type = 4
                        IF getInputMisMatch(SHA.input, Q-T2) == NULL
                                Bidder_preference = receipt.content.match_type + 7
                                insertWaitList(SHA, Q-T1, Bidder_preference)
                        END IF
                        ELSE
                                Desire = getInputMismatchMatch(SHA.input, Q-T2)
                                IF getMatch(SHA.input, Q-T2) == subsume
                                        Bidder_preference = receipt.content.match_type + 8
                                END IF
                                ELSE IF getMatch(SHA.input, Q-T2) == plugin
                                        Bidder_preference = receipt.content.match_type + 9
                                END IF
                                ELSE IF getMatch(SHA.input, Q-T2) == sibling
                                        Bidder_preference = receipt.content.match_type + 10
                                END IF
                                insertWaitList(SHA, Q-T1, Bidder_preference)
                        END IF
                END IF
        return done
        END IF
        END
```

Figure 64 (a): SmartDeal – Make Deal Phase

```
==========================================
ALGORITHM: SmartDeal (Query Behavior – SHA)
Input: Q-T1 [DL-code]
Output: done [Boolean]
==========================================
START
Message receipt = readMessage()
IF receipt.sender == SHA or UA && receipt.content.type == MakeDeal
        IF receipt.content.desire != NULL
                DA = chooseNearestDA(DA_List)
                sendMessage(DA, Query, receipt.content.desire)
        END IF
return done
END IF
END
```

Figure 64 (b): SmartDeal – Query Phase

```
==========================================
ALGORITHM: SmartDeal (Accept Deal Behavior – SHA)
Input: Q-T1 [DL-code]
Output: done [Boolean]
==========================================
START
IF DealList.size == 0 && CandidateList.count >= expected_size
        Candidate = selectBestCandidate(CandidateList, Q-T1)
        sendMessage(receipt.sender, AcceptDeal, Candidate, Q-T1)
return done
END IF
ELSE
        Message receipt = readMessage()
        IF receipt.sender == SHA or UA && receipt.content.type == AcceptDeal
                Candidate = getBidder(DealList, receipt.sender, Q-T1)
                insertCandidateList(Candidate)
                IF DealList.count == 0 && CandidateList.count >= expected_size
                        Candidate = selectBestCandidate(CandidateList, Q-T1)
                        sendMessage(receipt.sender, AcceptDeal, Candidate, Q-T1)
                END IF
        END IF
return done
END IF
END
```

Figure 64 (c): SmartDeal – Accept Deal Phase

```
=========================================
ALGORITHM: SmartDeal (Confirm Deal Behavior – UA/SHA)
Input: Q-T1 [DL-code]
Output: done [Boolean]
=========================================
START
IF Agent.this = UA && DealList.size == 0 && CandidateList.count >= expected_size
        Candidate = selectBestCandidate(CandidateList, Q-T1)
        sendMessage(Candidate, ConfirmDeal, Q-T1)
return done
END IF
ELSE
        Message receipt = readMessage()
        IF receipt.sender == UA && receipt.content.type == AcceptDeal
                Candidate = getBidder(DealList, receipt.sender, Q-T1)
                insertCandidateList(Candidate)
                IF CandidateList.count >= expected_size
                        Candidate = selectBestCandidate(CandidateList, Q-T1)
                        sendMessage(Candidate, ConfirmDeal, Q-T1)
                END IF
        END IF
        ELSE IF receipt.sender == SHA && receipt.content.type == ConfirmDeal
                IF CandidateList.count >= expected_size
                        Candidate = selectBestCandidate(CandidateList, Q-T1)
                        sendMessage(Candidate, ConfirmDeal, Q-T1)
                END IF
        END IF
return done
END IF
END
```

Figure 64 (d): SmartDeal – Confirm Deal Phase

6.10 Optimizing SmartDeal

In the previous section we detailed the *SmartDeal* algorithm in general. However, there are a few problems with the algorithm that still need to be addressed. In this section we first introduce each of these problems. After that we detail techniques to eliminate the problems thereby optimizing *SmartDeal*.

6.10.1 Problem of Make Deal Explosion

Make Deal Explosion is a phenomenon that may arise because deals that an agent makes can actually go on without terminating anywhere. This may occur in situations where the unmatched input parameters of a bidder *SHA* actually have very low correlation with the original desire of the *UA*. For an example if a *SHA* promises to give an output *car_information* but demands input *car_manufacturer* then it has very poor correlation with an *UA* that has desire {*car_information, rent_confirmation*}. However, the *SHA* might be picked up as a bidder since it partially satisfies the desire of the *UA*. Because of this the *SHA* will start believing that it can handle the user event and try looking for other bidder *SHAs* in vain.

To mitigate this problem each deal making agent (*UA* or *SHA*) tries to estimate the joint probability distribution of the given *UA* input and the unmatched bidder *SHA* output. In order to understand the acceptability of an estimated correlation the deal making agent keeps a *deal log* table. The *deal log* keeps a history of all successful deal bidder *SHA*s for a given desire. The *deal log* is a table of *N* 2-order tuple $\langle QSP, Pr \rangle$ where *Pr* is the joint probability of a *UA* desire *Q* and a past bidder *SHA's* input. Initially the *deal log* is empty. Hence, deal making agent does not discriminate and includes all bidder *SHAs* for a given desire. The joint probability is set to 1 for every bidder *SHA*. This continues till a set threshold *deal log* size is reached.

However, the deal maker agent keeps a timer and penalizes all those bidders that cannot accept the deal within the stipulated time. Penalization takes place in proportion to the popularity of the query (i.e. the number of times the same query is observed after the

first observation) and the failure rate (i.e. the number of failures versus query observations). The following model provides the penalization function:

$\Pr = \Pr - [Popularity \times Failure\ Rate]$ where *Popularity* and *Failure Rate* are:

$$Popularity_{\Delta T} = \left(\frac{\#\ Query\ Observations}{\Delta T_{query}}\right) \div \left(\frac{\#\ Total\ Query\ Observations}{\Delta T}\right)$$

$$Failure\ Rate_{\Delta T} = \left(\frac{\#\ Query\ Failure\ Observations}{\Delta T_{query}}\right) \div \left(\frac{\#\ Total\ Failure\ Observations}{\Delta T}\right)$$

and ΔT_{query}: Time since the query was first observed by the deal making agent;

ΔT: Time since any query was first observed by the deal making agent.

From the above mathematical model we can understand that the joint probability of a bidder *SHA* is seriously affected if it fails to accept a deal for a user query that is highly popular. However, the *SHA* is not that much affected in its early failures for a given query as it will be in its later failures. A *SHA* gets cancelled out of the *deal log* if it reaches a deviation lower than a threshold deviation σ_t from the mean joint probability of the *deal log*. In a similar way the deal making agent rewards a *SHA* in the *deal log* based on the following reward function:

$\Pr = (\Pr + [Popularity \times Success\ Rate]) \div Max\ Pr$ where Success Rate is:

$$Success\ Rate_{\Delta T} = \left(\frac{\#\ Query\ Success\ Observations}{\Delta T_{query}}\right) \div \left(\frac{\#\ Total\ Success\ Observations}{\Delta T}\right)$$

and *Max Pr* is the maximum joint probability in the *deal log*.

The *deal log* helps the deal making agent to understand: (a) whether a bidder *SHA* is reliable to deal with and (b) whether the bidder *SHA* is known before. For the first objective it checks the *deal log* to compute the current mean joint probability. After that it estimates

the joint probability of the current bidder *SHA* at hand and the given *UA* by using a semantic distance measure, called *SGPS measure*, proposed in [189]. The joint probability between two semantic parameters C_i and C_j is estimated as follows:

$\Pr(C_i, C_j) = Sim(C_i, C_j)$ where *Sim* is semantic similarity as per *SGPS* model.

Based on this estimation the joint probability between *n* parameters $C_1, C_1, \ldots C_n$ is estimated as:

$$\Pr(C_1, C_2 \ldots, C_n) = \frac{1}{\binom{n}{2}} \sum_i \sum_j \frac{Sim(C_i, C_j)}{2}$$

If the estimated joint distribution for a given unknown bidder *SHA* (i.e. a *SHA* that is not recorded in the *deal log*) has a deviation less that the current standard deviation $\sigma_{current}$ then it is considered as unreliable bidder. All unreliable bidders are put into a *waiting list* by a deal making agent so that it can come back to them for making deals if it cannot confirm any deal.

6.10.2 Problem of Starvation

Starvation is a situation where an agent keeps on waiting for a particular response so as to start a process. Hence, starvation can occur when a deal making agent has to keep on waiting for an *accept deal* response or an output response from a corresponding *SA*. One way to solve this problem has been discussed in the previous section. However, it may happen that the agent has to wait even when it has found the bidder *SHA* to be reliable. This may occur due to unprecedented network traffic overload or peer node overload where the bidder *SHA* lives. In such a starvation situation the deal making agent has no way to

understand whether the bidder *SHA* can finally accept the deal. We solve this problem in two different ways: (i) the waiting agent has a set timer as defined in the user preference, and (ii) the waiting agent estimates a wait time T_W after which it no longer waits. The first option is easy and trivial to implement. The second option is taken only when the first option is not available. A waiting agent computes the wait time as per the following model:

$$\mathrm{T_W} = \overline{\mathrm{T_W}} + \left(\left[Popularity \times Success\ Rate \times \frac{1}{1 + Criticality}\right] \times \overline{\mathrm{T_W}}\right)$$

where $\overline{\mathrm{T_W}}$: Average wait time experienced so far by the agent;

Criticality: Measure that indicates the importance of a user query in terms of desired service latency. Value range: [0, 1]

According to this estimation model an agent will wait more if the given user query is very popular and/or if the success rate of the responding agent is high. However, this can be seriously dampened by the *criticality* factor if the given user query happens to be a time critical query (example: online retrieval of medical information). The *criticality* is given either by the user or by the system designer based on some statistical estimation.

6.10.3 Problem of Confirm Deal Dilemma

Confirm Deal Dilemma is a problem that arises due to the uncertainty involved while looking for bidder *SHA*s by a deal making agent regarding the number of *SHA-clone* responses that it should expect. This is important because if the expected number of bidder *SHAs* is unknown to a deal making agent then it has no sound way to understand when it should send an *confirm deal* message to its bidders. As mentioned earlier, a confirm deal action is taken only when the *deal list* is empty. However, without knowing the size of the

deal list a priori the agent cannot be certain that the time is okay for confirming a deal by choosing the best bidder within the *candidate list*. It may happen that there are no bidder *SHAs* left but due to lack of information the deal making agent keeps on waiting. This seriously affects the overall service latency.

To solve this problem a *DA* is required to reply back to its querying deal making agent the number of *MC* matches it has observed for the query. If the number is *n* then there are *n BAs* to whom the query will be redirected. At the same time a *BA* is required to reply back to the same querying agent the total number of root *SHA-clones* that got matched in its *BA-directory*. If the number is *m* then the agent knows that there are *n* * *m* root *SHAs* that will communicate with it. Each root *SHA-clone* inside the *O-cluster space* of a specialized node is required to have a count of the number of descendant *SHA-clones* it has. Hence, if the i-th root *SHA-clone* reports the number as p_i then the deal making agent computes the maximum number of bidder *SHAs* (denoted as N_{bidder}) as:

$$N_{bidder} = \sum_{i}^{n \times m} p_i \ .$$

It is to be noted that N_{bidder} is an upper bound to the expected number of responses since root *SHA-clones* can share descendants within a more generalized *O-cluster space* topology. Therefore a deal making agent knows that it can receive a maximum of N_{bidder} *SHA-clone* responses and hence, does not wait for confirming a deal if the *deal list* is filled with this many entries.

However, as N_{bidder} is the upper bound therefore it may happen that the deal making agent wastes unnecessary time waiting for N_{bidder} number of entries in the *candidate list* and then finally reaches a timer expiry. Under such situation the deal making agent first checks

the *deal list* count. If the count is close to N_{bidder} by a chosen constant *k* then the agent checks if the *deal list* is static for a "long enough time". For that the agent uses the average time interval between two consecutive entries into the *deal list* (denoted by $\mu_{\Delta T}$). If the elapsed time is greater than $\mu_{\Delta T} + \sigma_{\Delta T}$ (where $\sigma_{\Delta T}$ is the standard deviation) then the agent drops its hope for getting new additions into the *deal list*.

6.11 Results

Evaluation of the *SMARTSPACE* platform has been implemented from two different perspectives:

- To understand the effectiveness of the *SmartCluster* algorithm for organizing *SHA-clones* into a *O-cluster space*.
- To understand the effectiveness of the *SmartDeal* algorithm for distributed *event-handling*.

For both the objectives we set up a simulation environment that typically represents a *SMARTSPACE* instance. We carried the simulation on a single desktop workstation (O/S: 32 bit Windows Vista) with CPU (Intel Core 2 Duo) cycle of 2.67 GHz and a RAM of 3 GB. We used JADE 3.7 as the underlying agent platform. The Java runtime environment was Eclipse Helios. The simulation framework consisted of two parts: (i) to develop a domain that is semantically represented as a set of ontologies, and (ii) to construct the service hosting P2P overlay on top of the JADE runtime platform.

For the first module of the simulation, called *OntoGenerator*, we synthetically constructed a set of ontologies and then *DL-encoded* them. The ontologies were constructed

as random acyclic graphs where we could control the sparseness in terms of a parameter called *Diversity Factor* (*DF*). We define *DF* as follows:

Definition 6.1: A *Diversity Factor* (*DF*) of a given set of ontologies S_O is the maximum probability with which two concepts C_i and C_j belonging to S_O have a mutual subsumption with each other. ■

From the definition we can understand that if the *DF* value is high chances are high as well that we get a very dense S_O that in itself signifies a very specialized domain in terms of breadth of individual ontologies. In contrast if the *DF* value is set low then S_O signifies a very generic and diverse domain. We also designed the simulation environment such that we have control over the size of S_O in terms of the total number of concepts. Thus, with a fixed *DF* if the S_O size is increased then there is high chance that the specificity of the domain will be increased in terms of depth of individual ontologies.

For the second module of the simulation, called *OverlayDesigner*, we designed a control mechanism for adjusting the total number of peer nodes and the total number of specialized nodes within the P2P network overlay. The total number of peer nodes was set to a constant of 10 for our purpose while the specialized node size was varied between 1, 4, and 6. We then implemented a random service generator that created a pool of 10,000 services by randomly selecting *DL-encoded* concepts from S_O and filling a particular service's *I-array* and *O-array*. We put a control over the minimum and maximum number of parameters that any *g-array* can have. For this present work we set that to [4, 5] for both *I-array* and *O-array*. Once services are generated they are kept in a pool for further implementation. After this stage is over services are assigned to the peer nodes randomly.

The random assignment is also controlled where we can pre-set the minimum and the maximum number of services per peer node. For this work it was set to [4, 4].

We also designed a *QueryShooter* module that generated a set of 100 *DQM* queries in random from S_O in the same way as how services were created. We set the minimum and maximum query parameter to [1, 3] for *Q-T2* and [5, 7] for *Q-T1*. The intention behind this set-up was to force *SHA*s to make at least one further deal during *event-handling* since they would not be fully satisfied by the *Q-T2* as they must have a minimum of 4 input parameters. On the other hand, *Q-T1* was kept above the output parameter interval since we wanted more than one *end service* to satisfy a given user query so as to ensure a more generic situation of *event-handling*.

At this point the simulation environment is ready for *SmartDeal* to take place and *JADE* runtime was booted up. *JADE* in turn loads *SMARTSPACE* on top of it. When *SMARTSPACE* gets loaded the services assigned to the peer nodes are converted into *SA*s. The conversion of services into *SA*s comes with a default *JADE* controlled interval of 0.0014 seconds on average. We term this interval *SA Birth-Interval* (*BI*). Since this interval is too un realistically fast for a given SOA-based system we put an extra *SMARTSPACE* control over the *BI* while keeping the *JADE BI* as a baseline for our observations. For this work we kept a *BI* of 1 second (which is also very fast and not so realistic) and another *BI* of 10 seconds that reasonably represents a very dynamic SOA-based system with high birth-rate (at least initially). The *QueryShooter* is not started before the *JADE* AMS agent (see chapter 2 for details) signals the creation of "enough" number of *SA*s in the system. We set this number from a range of 50 *SA*s to *1000 SA*s for different observations. The next sub-section

discusses the effectiveness of *SmartCluster* algorithm while the sub-section that follows it discusses the effectiveness of *SmartDeal* algorithm.

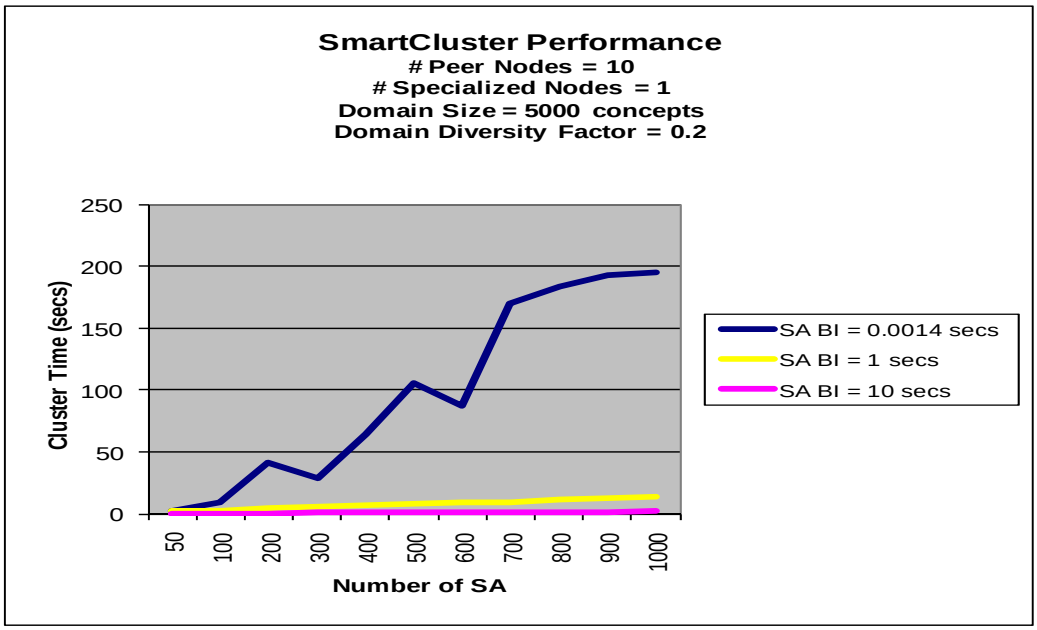


Figure 65: Effect of *BI* over *SmartCluster* Runtime Performance

6.11.1 SmartCluster Evaluation

The *SmartCluster* algorithm can be evaluated from two different perspectives: (i) accuracy, and (ii) runtime performance. Since *SmartCluster* is a version of *STC* from an algorithmic point of view hence, we do not discuss accuracy in this chapter. For detailed analysis and results on accuracy chapter 4 and chapter 6 can be consulted. In this section we are more interested in the runtime performance of *SmartCluster* since this gives us a direct insight on the ability of *SMARTSPACE* as a platform to perform distributed service discovery keeping the accuracy same as *STC*.

We first tried to understand the effect of *BI* over the *SmartCluster* algorithm. For that we chose the baseline *BI* and our own *BI* of 1 second and 10 seconds (figure 65). For this purpose we kept the domain size and the domain diversity to a constant of 5000 concepts and 0.2 respectively. The number of specialized nodes was kept to 1. We observed that for the *JADE* governed *BI* the overall performance was not that much promising with an average runtime of 98.13 secs (min: 2.562 secs for 50 SAs; max: 195.031 secs for 1000 SAs). The performance can mostly be attributed to a message overload per agent. This happened because the *SHA-clones* were very frequently born and sent to specialized nodes. This created a huge overhead on the sole *DA-BA* pair and also on the *SHA-clones* themselves since they have to communicate lot more frequently with new arrivals and between themselves and hence, consume a lot of *JADE* runtime resources. Furthermore it needs to be kept in mind that the experiment was conducted on a single machine with very limited memory and CPU cycle as compared to the scale in which the agents were created. However, when we relaxed the *BI* to 1 second we saw a drastic improvement in the performance with an average of 8 secs (min: 2.001 for 50 SAs; max: 14.21for 1000 SAs). We got an even better performance when we lowered the *BI* to a more realistic 10 seconds. The average runtime was 0.932 secs (min: 0.013 for 50 SAs; max: 1.964 for 1000 SAs).

We then observed the effect on the number of specialize nodes over *SmartCluster* to get an insight over the extent to which distribution of *O-cluster spaces* contributed to the overall performance. For this observation we kept the same domain configuration as before and also kept the more realistic *BI* of 10 seconds. The performance in this case was measured in milliseconds (figure 66). We saw that an increase in the *DA-BA* pair count

significantly released the overload over a single specialized node. For a size of 4 specialized nodes the average runtime performance dropped from 932 msecs (as noted earlier) to 96.396 msecs (min: 5.32 for 50 SAs; max: 179.994 for 1000 SAs). When we decreased the size

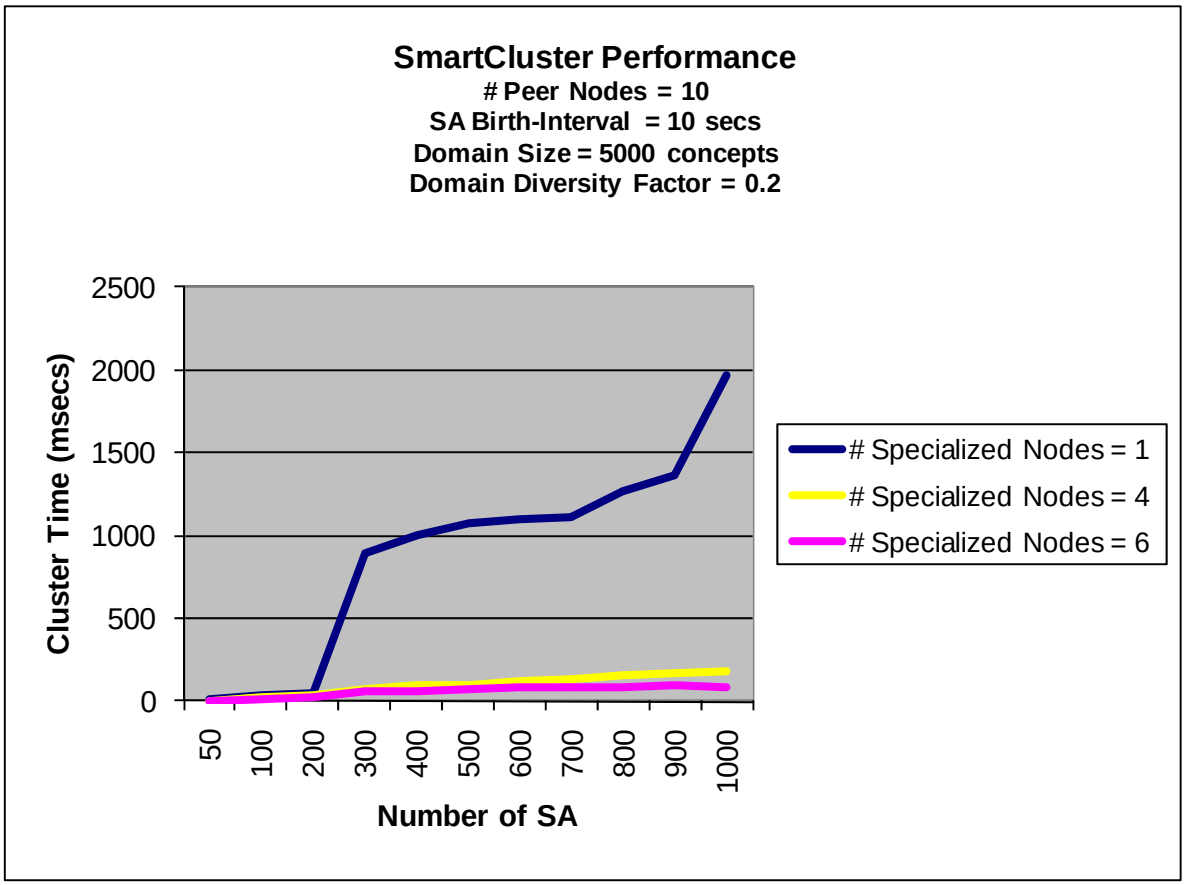


Figure 66: Effect of Specialized Node Count over *SmartCluster* Runtime Performance

further to 6 nodes we saw an average runtime of 60.405 msecs (min: 3.78 for 50 SAs; max: 83.892 for 1000 SAs). Based on the above observation we think that we do not need too many specialized nodes for improving the *SmartCluster* runtime. This is a positive sign since it is cost effective in terms of maintenance of specialized nodes. Also the number of mutual updates among *DA*s will not be much as well.

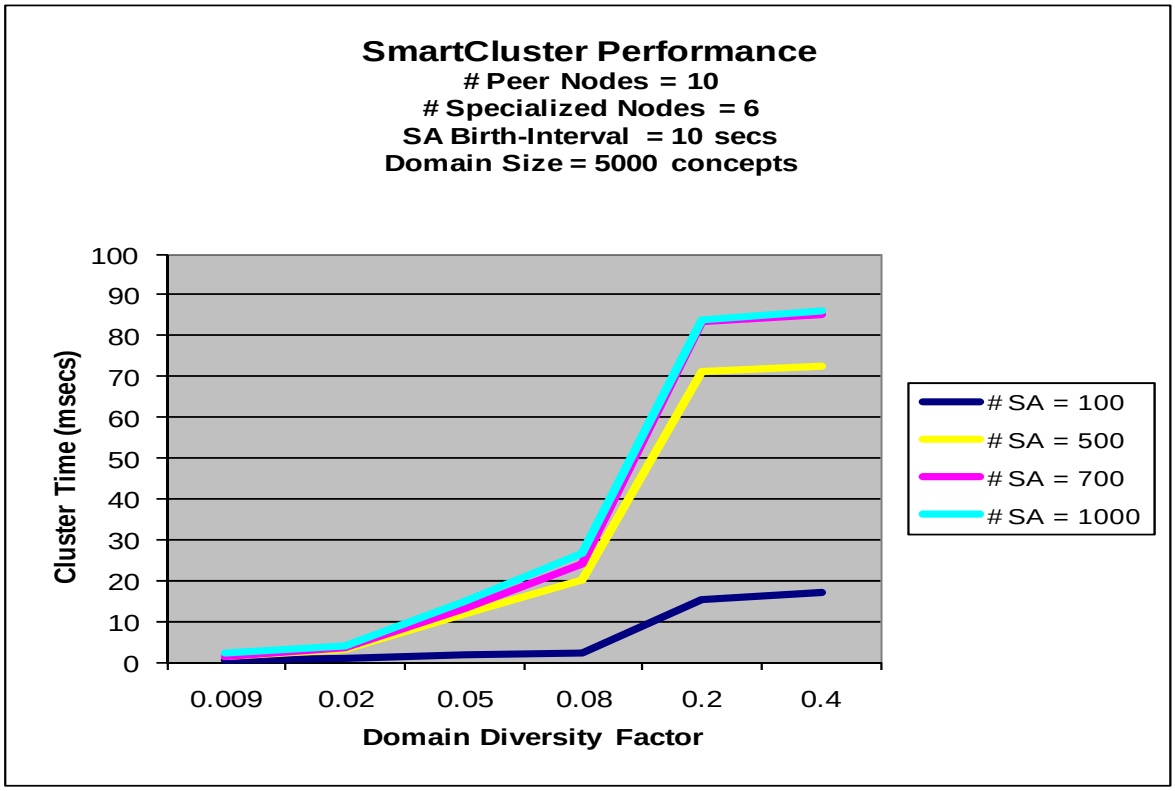


Figure 67: Effect of *DF* over *SmartCluster* Runtime Performance

The next experiment was to understand the effect of domain configuration over *SmartCluster*. For this purpose the number of specialized nodes was fixed to 6 and the *BI* was kept at 10 seconds. We first observed the effect of domain diversity in terms of *DF*. We chose 6 different *DF* values: 0.009, 0.02, 0.05, 0.08, 0.2, and 0.4. Figure 67 shows the *DF* effect over 4 different SA sizes. We observed that the runtime increases as the *DF* value increases. This is because as we increase the *DF* value the probability that two *SA*s having *g-relation* in terms of their *O-array* also increases. Hence, the inter-communication between *SHA-clones* and the *DA-BA* pair increases as well on an average. We then observed the effect of domain size in terms of number of ontology concepts over *SmartCluster*. We kept

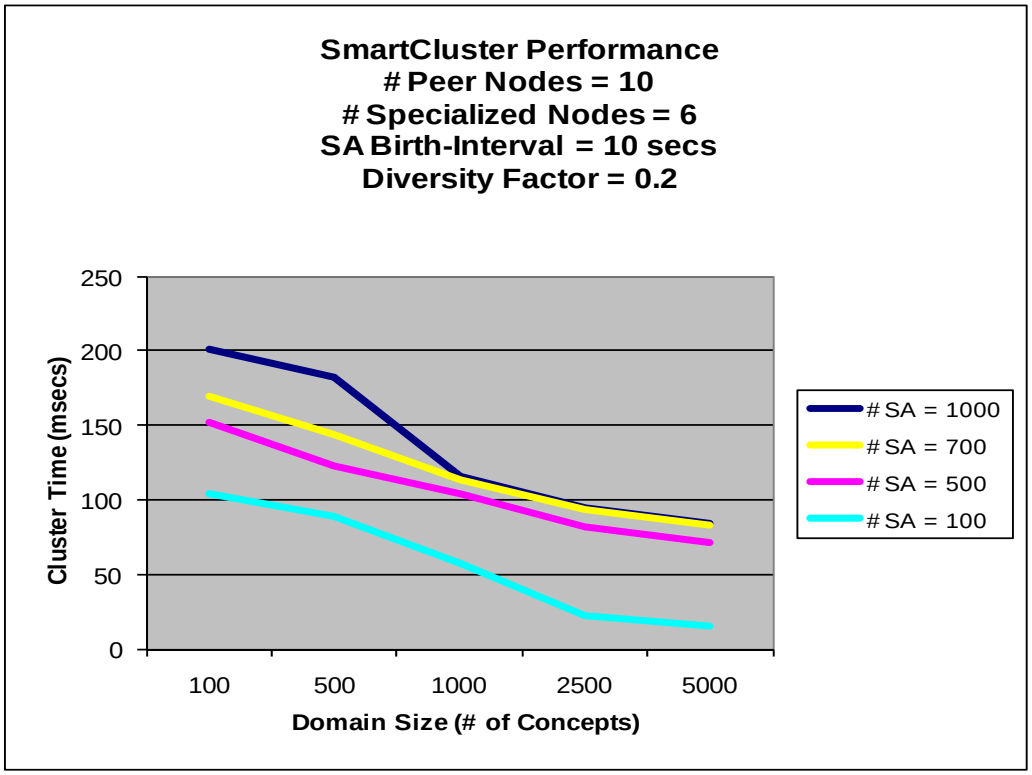


Figure 68: Effect of Domain Size over *SmartCluster* Runtime Performance

the *DF* value constant at 0.2. We chose the same sets of *SA*s as in the previous observation. We saw that the runtime improves when the number of concepts were increased from a minimum of 100 to a maximum of 5000 (figure 68). This is because as we increase the number of concepts keeping the *DF* constant we probabilistically achieve a domain that is diverse in terms of breath. Thus, we decrease the probability of two *SA*s to have their *O-arrays* in *g-relation* with each other. Therefore, the intercommunication overhead during *SmartCluster* is decreased as well on an average.

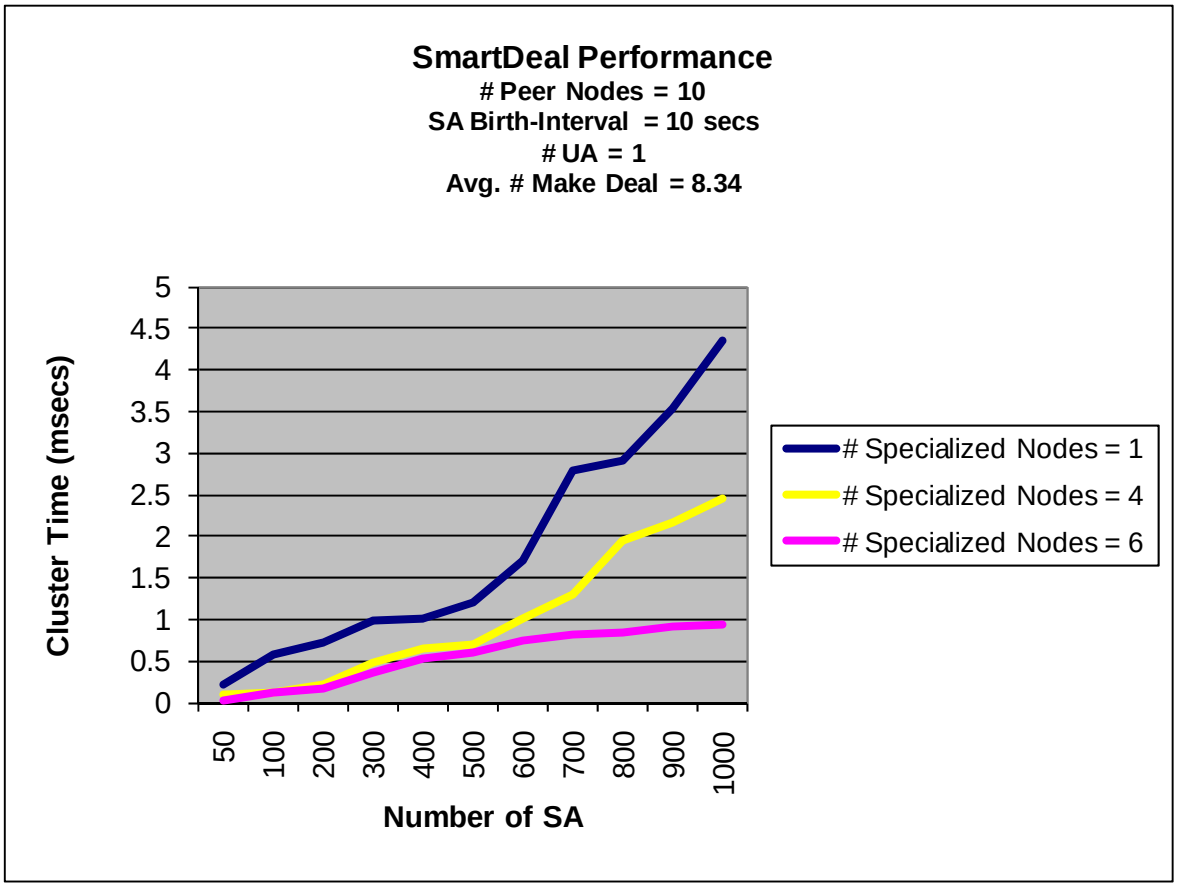


Figure 69: Effect of Specialized Node Count over *SmartDeal* Runtime Performance

6.11.2 SmartDeal Evaluation

The *SmartDeal* algorithm can be evaluated from several different aspects. However, in this work we chose to understand one of the most important aspects - performance. *SmartDeal* evaluation is essentially a complete evaluation of the *SMARTSPACE* platform. For the analysis in this section we manually chose a set of simulated query (generated by *QueryShooter* module) that gave us reasonably complex event-handling situations that involved service composition as such. This is important because we did not want to include cases where the problem just came down to a simple set of service discovery with no

composition taking place. The query set that we chose involved 8.34 average number of *make deal* behaviors per query by deal making agents. We used the same simulation environment that was used to test *SmartCluster*.

We first studied the effect of the number of specialized nodes on *SmartDeal* (similar to the test performed on *SmartCluster*). We kept the number of *UA* in the system to be just 1 - i.e. there is only one *UA* living in the entire system. We observed a some improvement on the runtime as we increased the number of specialized nodes from 1 to 6 (figure 69). The average runtime dropped from 1.818 msecs (min: 0.21 for 50 SAs; max: 4.34 for 1000 SAs) to 1.016 msecs (min: 0.105 for 50 SAs; max: 2.456 for 1000 SAs) when the specialized nodes were increased to 4 from 1. The average runtime dropped a bit more to 0.548 msecs (min: 0.03 for 50 SAs; max: 0.927 for 1000 SAs) when the specialized nodes were increased to 6. We did not see a significant drop in the runtime (unlike *SmartCluster*) because the *QueryShooter* is only started after the *SHA-clones* are clustered and it was only one Q-T1 query that had to be mapped over the *O-cluster spaces*. Therefore the number of specialized nodes only affects: (i) the *BA-directory* look-up time (since each *BA* will have less records to check if the system is more distributed over the specialized nodes) and (ii) the inter-agent communication between *SHA-clones* when a desire is mapped over the corresponding *O-cluster space*.

We then analyzed the effect of concurrency over *SmartDeal*. For that we made *SMARTSPACE* make clones a *UA* whenever *QueryShooter* created it. These clones represent a situation where several similar queries are put into the same system at the same time. We kept the range of clones from 0 (i.e. only 1 *UA*) to 999 (i.e. 1000 *UA*s). We took 4 different

system sizes (in terms of SA number). We observed a gradual but significant increase in the runtime which then grew rapidly as the number of clones were made more than 500 (figure 70). The significant deterioration in *SmartDeal* performance is mainly because of frequent starvation periods that a deal making agent on an average had to suffer. It is to be noted that

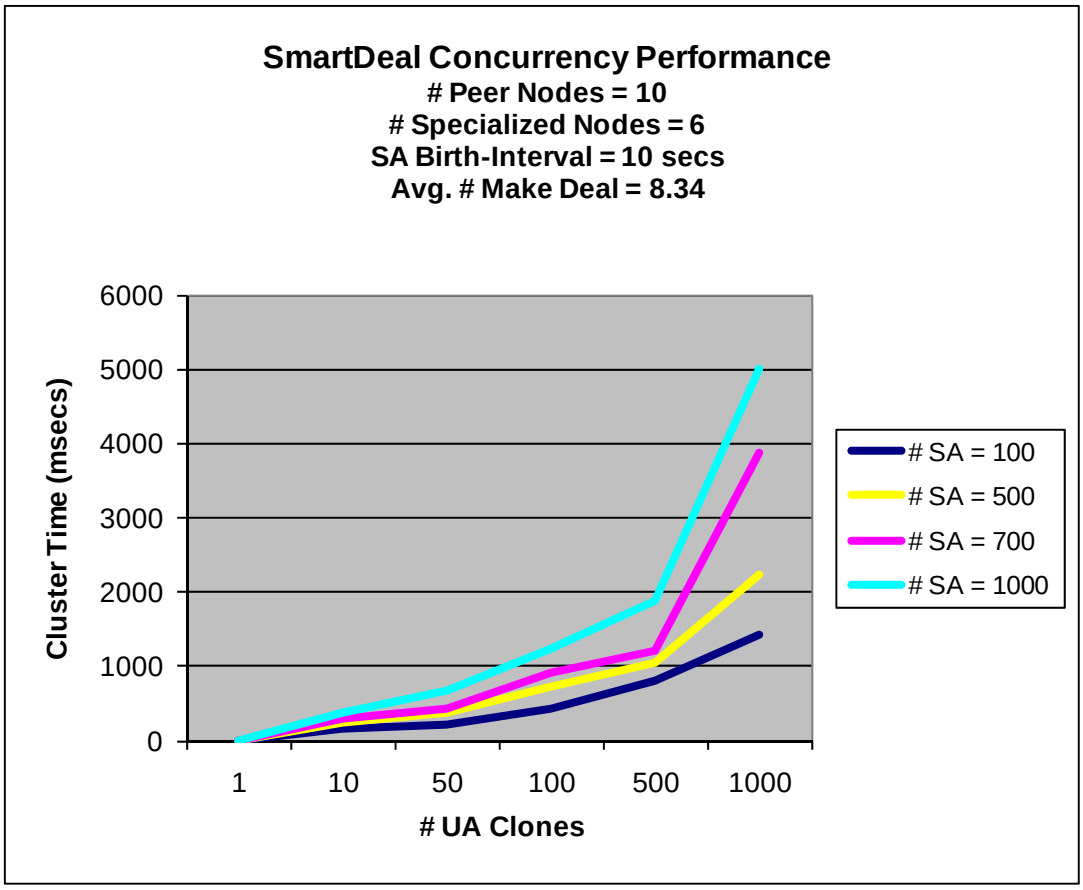


Figure 70: Effect of Concurrency over *SmartDeal* Runtime Performance

in our experiment we did not use any agent timer that might stop these starvation periods since we wanted to make sure that all the user events are handled (if can be handled). Another reason for not using timer is that we also wanted to understand the worst case handling time for a given query and hence, did not want to stop the process in between.

It is to be noted that *SmartDeal* was not evaluated in terms of *BI* since *QueryShooter* is started only after *SmartCluster* has been completed by *SMARTSPACE*. Hence, the *BI* would not have any effect as all the necessary *SA*s are allowed to be born before the evaluation starts.

6.12 Conclusion

SMARTSPACE is a novel distributed agent-based approach to the problem of *event-handling*. The platform has several features that we summarize below:

- **Stateful Asynchronous Service Composition**: In *SMARTSPACE* based event-handling a bidder *SHA* can store the corresponding *SA* output with the underlying AMS agent if it finds that the agent that confirmed the deal is busy. The busy agent can read the output once it is free. Also two deal making agents that are dealing with the same bidder *SHA* does not have to worry about syncing with each other. Messages are accumulated by the AMS and are forwarded to the bidder *SHA* as and when required. Thus, the underlying *JADE* runtime guarantees a truly stateful asynchronous service composition.
- **Failure-Tolerant Service Composition**: *SMARTSPACE* is a failure-tolerant service composition platform. This is because the agents that are involved in event-handling are mobile and hence, can move to other nodes the moment a particular node is sensed to be crashing.
- **Efficient Query Mapping**: *SMARTSPACE* provides a very efficient query mapping platform. Query mapping only requires a global abstract view of the *DA-directory* and the local taxonomic view of the *BA-directory*. This is complemented with fast

DL_encoding based lookups of these directories. Hence, the search space for user queries is narrowed down very quickly by the *DA* and the *BA*. Furthermore, query mapping only requires the *O-cluster space* (unlike the proposed *ALNet framework*) and hence, reduces a lot of extra maintenance as well as search overhead.

- **No Global Knowledge Requirement**: It is to be noted that within the a *SMARTSPACE* based system none of the agents has a complete understanding of the global system. In fact, each agent's belief is very limited to its own roles. Thus, *SMARTSPACE* is more flexible and adaptive under a very dynamic and uncertain system.
- **More Accurate Service Composition**: *SMARTSPACE* is sensitive to last-minute-changes that might occur within the system during event-handling. This is possible because service discovery and composition is truly integrated into one *event-handling* process in *SMARTSPACE*. Also agents are flexible to choose a better bidder *SHA* during the course of following a plan that did not include the bidder *SHA*. This flexibility is absent in centralized middleware dependent systems where the composition plan is strictly followed during runtime as planning is not integrated with the runtime binding.
- **Concurrent Service Composition**: *SMARTSPACE* can handle concurrency within the system with much less complexity and without having to re-compute the composition plan because of resource conflicts so common in concurrent systems. Efficient starvation policy helps *SMARTSPACE* to look-out for alternative plans on-the-fly.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Summary

In this dissertation we focused on the problem of service composition and elaborated on the various works that has been done in this area of study. We showed how reformulating the problem as *event-handling* problem can eliminate some of the computational hurdles that were innate in the more common formalization of the problem. We proposed a novel service category learning algorithm called *Semantic Taxonomic Clustering* (*STC*) that utilizes semantic descriptions of services written in languages like OWL-S. *STC* is an efficient way of organizing services into functional categories so as to prune search space during service discovery and also prune the search space for service composition. To compare semantic descriptions in an efficient way during *STC* we proposed the *g-subsumption* algorithm that leverages the proposed dynamic bit-based concept encoding algorithm called *DL-Encoding*. We then proposed two different frameworks for solving the *event-handling* problem - (i) *ALNet* framework and (ii) *SMARTSPACE* framework. While *ALNet* is a centralized asynchronous platform that is suitable more for relatively static systems *SMARTSPACE* is a distributed multi-agent based solution platform built over the *JADE* platform and is capable of scaling to large dynamic systems.

The contributions of the dissertation are summarized below:

- *DL-Encoding*: A Linear Time Description Logics (DL) Subsumption Testing Theory
- *g-subsumption*: An efficient semantic service matchmaking algorithm
- *STC*: A novel and efficient semantic service category learning algorithm

- *DQM*: A novel *DL-Encoding* based query model for efficient query mapping
- *Notability Theory*: A formal theory for agent-based cognitive modeling of events
- *ALNet*: A novel service dependency graph based centralized event-handling platform
- *SMARTSPACE*: A novel multi-agent based distributed event-handling platform

7.2 Future Work

In this section we discuss the limitations of some of the current work and the directions in which we plan to move on regarding further research. We identify three main areas that can be improved significantly: (i) *DL-Encoding* theory, (ii) *SMARTSPACE* analysis, and (iii) Context-aware event-handling.

7.2.1 *DL-Encoding* Theory Extension

DL-Encoding is currently applicable to the DL language $\mathcal{ALC}_{\mathcal{R}^c}\mathcal{H}$. However the language does not support some important DL constructs such as number restrictions. Also some of the DL properties such as concept unstatisfiability and rule satisfiability need to be further analyzed for the proposed theory. We think that in general *DL-Encoding* can be theoretically extended further that might reveal more interesting DL properties.

Moreover, currently *g-subsumption* that is based on *DL-Encoding* is only over the *I-array* and the *O-array*. The pre-condition and the effect features of service descriptions have been largely ignored in this work. We would want to extend the *g-subsumption* algorithm to include the fuller service feature set. This also includes the *DQM* query model that must be extended to include user defined constraint in a succinct format.

7.2.2 *SMARTSPACE* Analysis

SMARTSPACE is a relatively new project and has a lot of scope for further evaluation and analysis. This is especially so with respect to the different types of important analysis that can be done over the *SmartDeal* algorithm. In the near future we plan to observe and understand the exclusive effects of having: (i) starvation policy, (ii) make deal explosion policy, and (iii) confirm deal dilemma policy on *SmartDeal* accuracy performance. Accuracy measurement is challenging in the current simulation environment since the system dynamics is totally unknown. We plan to record this dynamics during an *event-handling* span and later compare the composition given by *SmartDeal* with the actual optimal composition under the last recorded system state. This will allow us to understand the number of bidder *SHAs* that are ignored by *SmartDeal* and also how much that affects the overall quality of the composition.

We also plan to understand the overall dynamics of *SmartDeal* when *SAs* are allowed to die as well. It is to be noted that in chapter 6 we only had the *BI* parameter but we also need to have a *DI* (*death-interval*) parameter into the simulation as well. It will be very helpful to analyze both the accuracy as well as the computational efficiency of *SmartDeal* in an environment that is under constant change even during the *event-handling* process. In our current analysis we have assumed that *event-handling* starts only when the system attains a stability (i.e. *SmartCluster* is executed and the system is static). Also, inducing node failures into the simulation can help us to understand the resiliency of *SMARTSPACE*.

7.2.3 Context-Aware Event-handling

At present we have not taken into consideration the complex effects of user events that demand context-awareness from the service composition platform. In the use case example Chris wants to rent a car for going to Chicago from Kansas City. He is supposed to stay there for a week and fly back to Kansas City. Imagine a situation where the underlying compound *DQM* query is fed into *SMARTSPACE*. Two different *UA*s are going to deal very independent of each other and create two independent event-handling processes. However, if the car renting *UA* fails then there is no way to alert the flight booking *UA* that its service may not be required any more. The problem is in the lack of understanding between the *UAs* that even though they have simplified the initial compound query into simple queries yet they are not independent and have an innate temporal relation. Such understanding requires context analysis of the original query at the time of breaking it into simpler queries. We understand that the problem of context-awareness can be understood from two different perspectives: (i) a priori context modeling and (ii) dynamic context-learning.

7.2.3.1 A Priori Context Modeling

Context analysis is an extremely difficult problem. This is even so because formalization of context is not easy. Previous works (such as Context ToolKit [238], CoBrA [239 - 240]) has described context from a broad perspective. According to CoBrA context implies the understanding of: (i) location, (ii) location environmental attributes (i.e. noise level, temperature, light intensity, motion), (iii) location entities (i.e. people, devices, objects, software agents). SOUPA [241] also capitalizes on such a conception. In other

words, context is mostly a location-centric concept in these approaches. In the words of Dey [242] a context is defined as: "…… *any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.*" This definition of context implies that context characterizes: (i) state of entity, (ii) activity of entity – provided we accept that *situation* consists of the state and the activity. The definition also implies that context should be useful to the world in some ways – specifically when the world entities interact among themselves. Even though many researches in the field of context-awareness have accepted this definition as a standard guideline yet there is a certain degree of ambiguity in it. We enumerate them as follows:

a) If we assume that *situation* means the state and activities of entities at a particular point of time then *context* is just the same as the system state associated with an entity. This is problematic because this does not really tell us: (i) whether the state information is useful at all, (ii) when that state information can be useful for other agents, and (iii) how the state information will be utilized by other agents. The problem leads to inefficient resource utilization as agents will always have to collect or report context information to other agents either via a P2P communication infrastructure or via a global management framework.

b) The second important drawback in this definition is that we cannot properly distinguish between when state information is a required input to other agents and when the information is something additional (contexts are normally considered as additional *useful* information). This is important as input is necessary and sufficient information for

an agent to behave while context is not. Context influences the mode of behavior of an agent in three ways: (i) changes the format of an output, (ii) fine-tunes the granularity of an output, and (iii) suppresses (or over-rides) the output. However, it cannot change the behavior itself. Let us illustrate this example with a toy use case:

John wants to read the morning news every day. Normally he wakes up and enters the hall where a huge 46 inch display board is set up. The intelligent space senses his presence and displays the news. However, on a particular day John is late for office and instead wants to listen to the news in this car audio system. The intelligent space is aware that John is in his car.

In this example, *John's identity* is an input to the agent that provides news; while *John's location* is the context. Both are the states of the entity John. However, the agent providing the service will be working fine when it detects John in the car and displays the news in the hall. The context helps it to fine-tune the output *morning news* into an audio-only format when John is in his car. But it cannot change the behavior of the agent itself by turning it into, say, a music providing agent. Thus, we need a formalism by which we can clearly distinguish between context and input information.

c) The concept *situation* itself adds up to the ambiguity. According to Endsley et al. [243] situation-awareness is defined as: "*the perception of elements in the environment within a volume of time and space, the comprehension of their meaning, and the*

projection of their status in the near future"[1]. This definition implies that: (i) situation is a conglomeration of several entity states, (ii) the states collective has an interpretation for one or more agents, and (iii) a *situation gradient* may be extrapolated for determining future situations within the system. In other words, situation can be viewed as a logical constitution of several contexts that has some implication for now or future. Hence, the context definition may be considered cyclical from a conventional notion of *situation* in the community

d) Although there has been a lot of attention regarding the representation of context in pervasive system frameworks [239 - 241] context is yet to be formalized mathematically. This is necessary because it provides a computational basis for any framework to distinctively understand which state is a context to which agent and how that is related to the agent's behavior. In other words, the framework does not need explicit apriori definition of what is context for a system. It can also utilize the resource efficiently by not collecting garbage state information but only those that constitute the context. Lack of formalism in existing ontological frameworks prevents new instances of context from getting automatically classified into the framework.

e) In many works related to policy management in pervasive systems contexts (specifically context-conditions) are used for framing policies [244]. This has several advantages in a system where agents cannot be known apriori. If context definition

[1] Endley's definition is considered the most standard one. Other definitions can be seen as a generalized derivation of this definition.

statically depends upon agent states as well then we cannot in reality frame the policies. Thus, context definition should be dynamic in the sense that what is context should be determined by the framework runtime. Based on that the framework should be able to automatically generate and/or suggest policies that are required for managing the agents.

Based on the above discussions we can see that the *notability theory* introduced in chapter 6 can be integrated very easily with *SMARTPACE* in order to resolve some of the problems mentioned. We can extend the *notability theory* to model context space as follows:

Definition 7.1: An *context vector space* of an event ε_i^a (denoted as $\overline{C_{\varepsilon^a}}$) is the set X s.t. $(X \subseteq \overrightarrow{\varphi_{\varepsilon_i^a}}) \wedge (\forall x \in X, \exists A \in U_a^C;\ \tau^x \xrightarrow{\vDash} A)$ where τ^x is the transition from the initial state value to the final state value of the vector element x and U^C is the set of context axioms. ■

The definition implies: (i) the context can be expressed in terms of the state of an event, (ii) there must be an agent a that notes the event and to which the context is meaningful, (iii) there is a set of context axioms U^C that comprises of first-order axioms written in the event-context_condition-action format, and (iv) all the vector elements constituting the context should contribute in satisfying at least one context axiom. In the given toy use case the context condition can be simplified as: *if person.location:= car then output.format:= audio*. In this case the vector element *person.location* should undergo a value transition from *!car* to *car* so as to trigger the context axiom.

As context space is expressed in terms of event states hence, we can design an extended *CAOFES* semantic framework where context can be relationally connected to an event's state. This helps reasoning about context by understanding the state vector. Moreover, newer states can be verified whether they are context to any agent by

understanding whether the states trigger off any of the agent's context axioms. The separation of context axiom from role axioms is significant because that helps the reasoner to distinguish whether a state is an input (it is so if it satisfies a role axiom) or whether the state is the context. In the toy use case the state *person.ID:= John* is an input as it triggers the role axiom: if *person.ID:= John* then *output:= morning news*.

However, context axioms are not easy to frame within an open, dynamic and evolving system. This is especially true within an SOA based system where the type of user events and services is very difficult to estimate. Hence, a priori definitions of crisp context axioms are not possible. Under such circumstances we think that we need to understand the significant (and widely ignored) difference between a *context space* and *context* itself. We think that context should be modeled as agent intention (and a corresponding intention utility function) rather than a pre-defined set of axioms. For a particular agent intention there is a well-defined *context space* that needs to be carved out on the run by agents. The overall objective of all cooperating agents is to maximize the intention utility function of an agent that is served. For an example, if the agent intention is to do an indoor work then the corresponding context space can be room temperature, light intensity, indoor noise, resource availability, etc. In contrast if the agent intention is to do an outdoor activity then the corresponding context space can be weather condition, resource availability, time scheduling, etc. Based on such identification generic context axioms can be triggered and then through techniques such as reinforcement learning we can narrow down the axioms to more specific ones. This is done by eliminating unnecessary context space vectors and replacing generic vectors with more specific vectors. For an example, it may be found that

for a cooking activity intention noise parameter is not required while light intensity can be more specialized to kitchen light intensity.

7.2.3.2 Dynamic Context Learning

Context learning is a situation where the server agent has no prior knowledge of what can be the intention of an agent that requires service. This may happen when an user event is very new to the system. Estimating agent intention is an extremely difficult and not so well formed problem. There are several dimensions to this problem:

- **Context Analysis of Query**: Sometimes the true intention can be uncovered from the query term semantics collectively. For an example, if a compound query desire is framed as {<*rental car info, confirmation*>, <*flight info, confirmation*>} then collectively the two sub parts of this compound query are semantically related to travel. Now a travel intention itself has an innate temporal implication and the corresponding utility function cannot be optimized if the temporal order is not maintained. This understanding may actually lead to a joint collaboration between the two *UAs* (and their bidder *SHAs*) in ensuring that the utility function is satisfied. Thus, the flight booking *UA* understands that there is no way that the utility can be satisfied if it does its job alone. However, formalizing the utility function itself is a problem as hard as the problem of context-learning.
- **Context Analysis of Past Behavior**: Lot of times it is necessary to understand the past behavior of a service requesting agent in order to estimate its current intention. The temporal order and the behavior semantics can actually indicate something very different

in comparison to intention estimation without consideration of past history. This is because many times intention has both a global as well as a local implication. For an example, if an agent says "*I am hot*" then there can be at least two conflicting semantics of the term *hot* - (i) temperature and (ii) taste. If such semantic conflicts is not resolved then there can be two parallel event-handling process (one is *drink providing* and the other is *temperature lowering*) in execution within the system. However, investigating past actions might actually tell the drink providing *UA* that the agent has eaten something before this request. *Eating* has a semantic interpretation for this *UA* while for the temperature controller *UA* it has no interpretation at all. Thus, past behavior can resolve such semantic ambiguities.

- **Context Analysis of Current User State**: The user profile cannot be assumed to be static all the time. The user can move to some other place during the span of the event-handling process (for an example Chris can move to his mom's city in Manhattan after requesting car rental from Kansas City) or may change his preference (Chris does not want an SUV any more) or his personal situation (Chris's meeting at Chicago has now been moved to Detroit). In such a situation the *UA*s working for Chris need to update themselves proactively based on an understanding of all those context space variables that directly affect the intention utility (such as location of rental car pick-up).

Context-learning is going to be a long-term research study for the ongoing *SMARTSPACE* project. It is a relatively less ventured area in all its forms when compared to context modeling. To sum up the differences between these two problems we can say that while context modeling is more worried about the question: "Does the current situation

satisfy the context axioms for an agent?" context learning, on the other hand, meddles with the even more difficult question: "What can be the context axioms of an agent at this moment?".

We hope that we will get some valuable insights into these two problems and try to understand their implication in the larger study of distributed context-aware service composition.

REFERENCES

[1] Berners-Lee, T. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide*. Harper, San Francisco, 1999.

[2] Greenfield, A. *Everyware: the dawning age of ubiquitous computing.* New Riders, Berkeley, 2006

[3] Microsoft Corporation. *The Component Object Model Specification*. October 1995, Version 0.9.

[4] Object Management Group. *The Common Object Request Broker Architecture (CORBA) core specification*. December 2002, Version 3.0.

[5] OASIS. *Reference Architecture for Service Oriented Architecture Specification*. April 2008, Version 1.0

[6] Nickull, D. *Service Oriented Architecture*. Adobe Systems Incorporated. Available from http://www.adobe.com/jp/enterprise/pdfs/Services_Oriented_Architecture_from_Adobe.pdf (accessed 12 April 2011). 2005

[7] Papazoglou, M. and Georgakopoulos, D. Service-Oriented Computing. *Communications of the ACM*, 46 (10), 2003, 25–28.

[8] Bellwood, T., et al. *UDDI Spec Technical Committee Draft*. 2004, Version 3.0.2. Available from http://www.uddi.org/pubs/uddi_v3.htm.

[9] ebXML Technical Architecture Team. *ebXML technical architecture specification*. February 2001, Version 1.0.4. Available from http://www.ebxml.org.

[10] Box, D., et al. *Simple Object Access Protocol (SOAP)*. 2001, Version 1.1. Available from http://www.w3.org/TR/SOAP/ (2001).

[11] Laliwala, Z. and Desai, A. Policy-based Services Aggregation in Grid Business Process. *Annual IEEE India Conference* (*INDICON)*, India, 2009.

[12] Levina, O. and Stantchev, V. Realizing Event-Driven SOA. *4th International Conference on Internet and Web Applications and Services*, Italy, 2009

[13] Ye, C. and Jacobsen, H-A. Event Exposure for Web Services: A Grey-box Approach to Compose and Evolve Web Services. *The Smart Internet*, *Springer LNCS*, 6400, 2010.

[14] Taylor, H., Yochem, A., Phillips, L., and Martinez, F. *Event-Driven Architecture: How SOA Enables the Real-Time Enterprise*. Addition-Wesley Professional, 2009

[15] Luckham, D.C. and Frasca, B. *Complex event processing in distributed systems*. Stanford University, Technical Report, 1998.

[16] Mühl, G., Fiege, L., and Pietzuch, P. *Distributed Event-Based Systems*. Springer, 2006.

[17] Chandy, M.K. Event-Driven Applications: Costs, Benefits and Design Approaches. *Gartner Application Integration and Web Services Summit*, Orlando, USA, 2006.

[18] Chinnici, R., et al. *Web Services Description Language (WSDL)*. 2001, Version 1.2. Available from http://www.w3.org/TR/wsdl (2001).

[19] Andrews, T., et al. *Business Process Execution Language for Web Services (BPEL4WS)*, May 2003, Version 1.1. Available at http://www-106.ibm.com/developerworks/webservices/library/ws-bpel.

[20] Hewlett-Packard Company. *E-speak Architectural Specification*. Version A.0, January 2001. Available at http://www.hpl.hp.com/-personal/Alan Karp/espeak/version3.14/-Architecture 3.14.pdf

[21] Chou, D., deVadoss, J., Erl, T., Gandhi, N., Kommalapati, H., Loesgen, B., Schittko, C., Wilhelmsen, H., and Williams, M. *SOA with .NET & Windows Azure: Realizing Service-Orientation with the Microsoft Platform*. Prentice Hall, Boston, 2010.

[22] Chappel, D. *Understanding BizTalk Server 2006*. Microsoft White Paper, August 2005.

[23] Sun Microsystem. The Sun Open Net Environment (Sun ONE). *System News,* 36(1), 2001.

[24] Oracle Corporation. Oracle Dynamic Services User's and Administrator's Guide. 2000, Version 9.0.1

[25] Budinsky, F., DeCandio, G., Earle, R., Francis, T., Jones, J., Li, J., Nally, M., Nelin, C., Popescu, V., Rich, S., Ryman, A., and Wilson, T. WebSphere Studio overview. *IBM Systems Journal* 43 (2), 2004, 384–419.

[26] VerticalNet Solutions. *OSM Platform Overview*. December 2000, Version 1.0. Available at http://adam.cheyer.com/docs/OSMOverview.pdf

[27] Shrivastava, S., Bellissard, L., Fliot, D., et al. A workflow and agent based platform for service provisioning. *Proceedings of the 4th IEEE/OMG International Enterprise Distributed Object Computing Conference(EDOC 2000)*, Makuhari, Japan, 2000.

[28] Casati, F., Ilnicki, S., and Jin, L. Adaptive and dynamic service composition in EFlow. *Proceedings of 12th International Conference on Advanced Information Systems Engineering(CAiSE)*, Stockholm, Sweden, 2000.

[29] Turner, M., Budgen, D., and Brereton, P. Turning software into a service. *IEEE Computer*, 36(10), October 2003.

[30] Berners-Lee, T., Hendler, J., and Lassila, O. The Semantic Web. *Scientific American*, May 2001, 29-37.

[32] Berners-Lee, T. and Miller, E. The Semantic Web lifts off. *ERCIM News*, 51, 2002, 9-10.

[33] Bray, T., Paoli, J., Sperberg-McQueen, C. M. and Maler, E. *Extensible Markup Language (XML)*. October 2000, Version 1.0 (second edition), W3C recommendation 6. Availble at http://www.w3.org/TR/2000/REC-xml-20001006.

[34] Lassila, O. and Swick, R. R.. *Resource Description Framework (RDF) model and syntax specification*. February 1999, Version 1.2, W3C recommendation. Available at http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[35] Brickley, D. and Guha, R. V. *RDF Vocabulary Description Language 1.0: RDF Schema*. February 2004, Version 1.0. Availeble at http://www.w3.org/TR/rdf-schema/.

[36] Hendler, J. and McGuinness, D.L. The DARPA Agent Markup Language. *IEEE Intelligent Systems*, 15 (6), 2000, 67-73.

[37] Fensel, D., Horrocks, I., van Harmelen, F., Decker, S., Erdmann, M. and Klein, M. OIL in a nutshell. *Proceedings of the European Knowledge Acquisition Conference (EKAW-2000)*, 1937, LNAI, Springer-Verlag, October 2000.

[38] Horrocks, I. DAML+OIL: A Description Logic for the Semantic Web. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 25(1), 2002, 4-9.

[39] Baader, F., Horrocks, I., and Sattler, U. *Handbook of Knowledge Representation* (*Chapter 3 – Description Logics*). Elsevier, 2007.

[40] Dean, M. et al. *OWL Web Ontology Language 1.0 reference*. July 2002, Version 1.0. Available at http://www.w3.org/TR/owl-ref/.

[41] Horrocks, I. The FaCT system. In *Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98*, 1397, LNAI, Springer-Verlag, 1998, 307-312.

[42] Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., and Katz, Y. Pellet: A Practical OWL-DL Reasoner. *Journal of Web Semantics: Science (Services and Agents on the World Wide Web)*, 5(2), 2007.

[43] Racer Systems GmbH & Co. K. RacerPro Reference Manual. December 2005, Version 1.9. Available at http://www.racer-systems.com/products/racerpro/reference-manual-1-9.pdf

[44] Bechhofer, S., Horrocks, I., Goble, C., and Stevens, R. OilEd: A Reasonable ontology editor for the semantic web. *Proceedings of the 2001 Description Logic Workshop (DL 2001)*, CEUR, 2001, 1–9.

[45] Knublauch, H., Fergerson, R.W., Noy, N.F., and Musen, M.A. The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. *International Semantic Web Conference* (*The Semantic Web*), 3298, 2004, 229-243.

[46] Fensel, D., Lausen, H., Polleres, A., de Bruijn, J., Stollberg, M., Roman, D., and Domingue, J. *Enabling Semantic Web Services: The Web Service Modeling Ontology*. Springer-Verlag, New York, 2006.

[47] Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., and Fensel, D. Web service modeling ontology. *Applied Ontology*, 1(1), 2005, 77–106.

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

[49] Lausen, H., de Bruijn, J., Polleres, A., and Fensel, D. WSML - A Language Framework for Semantic Web Services. *Rule Languages for Interoperability*. W3C Rules Workshop, Washington DC, 2005.

[50] Burstein, M., Hobbs, J., Lassila, O., Mcdermott, D., Mcilraith, S, Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., and Sycara, K. *OWL-S: Semantic Markup for Web Services*. November 2004, W3C recommendation. Available at http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/

[51] Martin, D., Burstein, M., Mcdermott, D., Mcilraith, S, Paolucci, M., Sycara, K, Mcguinness, D.L., Sirin, E., and Srinivasan, N. Bringing Semantics to Web Services with OWL-S. *World Wide Web*, 10(3), 2007, 243–277.

[52] Martin, D. et al. *DAML-S (and OWL-S) 0.9 draft release*. May 2003, Version 0.9. Available at http://www.daml.org/services/daml-s/0.9/.

[53] Battle, S., Bernstein, A., Boley, H., Grosof, B., Gruninger, M., Hull, R., Kifer, M., Martin, D., Mcilraith, S., Mcguinness, D.L., Su, J., and Tabet, S. *Semantic Web Services*

Framework (SWSF) Overview. September 2005, Version 1.0, W3C member submission. Available at http://www.w3.org/Submission/SWSF/.

[54] Gruninger, M. and Fox, M. Methodology for the design and evaluation of ontologies. *IJCAI'95, Workshop on Basic Ontological Issues in Knowledge Sharing*, Montreal, Canada, 1995.

[55] Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M-T, Sheth, A, and Verma, K. *Web Service Semantics – WSDL-S*. November 2005, Version 1.0, W3C member submission. Available at http://www.w3.org/Submission/WSDL-S/

[56] Patil, A., Oundhakar, S., Sheth, A., and Verma, K. METEOR-S Web Service Annotation Framework. *Proceedings of the Thirteenth International World Wide Web Conference (WWW2004)*, New York, USA, 2004.

[57] Farell, J. and Lausen, H. *Semantic Annotations for WSDL and XML Schema (SAWSDL)*. August 2007, Version 1.2. Available at http://www.w3.org/TR/sawsdl/.

[58] Foster, H., Uchitel, S., Magee, J., and Kramer, J. WS-Engineer: A Model-Based Approach to Engineering Web Service Compositions and Choreography. *Test and Analysis of Web Services*, Springer Berlin, Heidelberg, 2007.

[59] Bultan, T., Fu, X., Hull, R., and Su, J. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. *Proceedings of the Twelfth International World Wide Web Conference (WWW'2003)*, Budapest, Hungary, 2003.

[60] Yang, H., Zhao, X., Qiu, Z., Pu, G., and Wang, S. A Formal Model for Web Service Choreography Description Language (WS-CDL). In *International Conference on Web Service (ICWS)*, Chicago, USA, 2006, 893–894.

[61] Boh, W. F., Soh, C., and Yeo, S. Standards development and diffusion: a case study of RosettaNet. *Communications of the ACM*, 50 (12), 2007, 57–62.

[62] http://www.swift.com

[63] http://www.hl7.org/implement/standards/index.cfm?ref=nav

[64] Perttunen, M., Jurmu, M., and Riekki, J. A QoS Model for Task-Based Service Composition. *4th International Workshop on Managing Ubiquitous Communications and Services (MUCS 2007)*, Munich, Germany, 2007.

[65] Sousa, J., Poladian, V., Garlan, D., Schmerl, B., and Shaw, M. Task-based Adaptation for Ubiquitous Computing. *IEEE Transactions on Systems, Man, and Cybernetics*, 36(3), 2006. 328-340.

[66] Rao, J. and Su, X. A survey of automated web service composition methods. *Proceedings of the 1st International Workshop on Semantic Web Services and Web Process Composition (SWSWPC2004),* San Diego, USA, 2004.

[67] Narayanan, S. and Mcllraith, S. A. Simulation, Verification and Automated Composition of Web Services. *Proceedings of the 11th International. WWW Conference,* Honolulu, 2002.

[68] Benatallah, B., Dumas, M., Sheng, Q., and Ngu, A. Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services. *IEEE International. Conference on Data Engineering*, San Jose, 2002.

[69] Tosic, V., Mennie, D. and Pagurek, B. On Dynamic Service Composition and Its Applicability to E-business Software Systems. *Workshop on OO Business Solution (ECOOP)*, Budapest, Hungary, 2001.

[70] Martello, S. and Toth, T. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, USA, 1990.

[71] Tsesmetzis, D., Roussaki, I, and Sykas, E. QoS-aware service evaluation and selection. *European Journal of Operation Research*, 191, 2008, 1101 – 1112.

[72] Alrifai, M. and Risse, T. Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition. *Proceedings of the 18th International World Wide Web Conference*, Madrid, 2009.

[73] Casati, F. and M. Shan. Dynamic and Adaptive Composition of e-Services. *Information Systems*, 26(3), 2001, 143 – 162

[74] Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., and ter Hofstede, A.H.M. Formal semantics and analysis of control flow in WS-BPEL. *Scientific Computing Program*, 67(2-3), 2007, 162–198.

[75] Brogi, A. and Popescu, R. Towards Semi-Automated Workflow-based Aggregation of Web Services. *Proceedings of 3rd International Conference on Service Oriented Computing (ICSOC'05)*, Amsterdam, Netherlands, 2005.

[76] Tut, M. T. and Edmond, D. The Use of Patterns in Service Composition. *Proceedings of the 1st Workshop of Web Services, e-Business and the Semantic Web*, Toronto, Canada, 2002.

[77] Wu, D., Parsia, B., Sirin, E., Hendler, J. and Nau, D. Automating DAML-S Web Services Composition Using SHOP2. *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, Florida, USA, 2003.

[78] Mcilraith, S. and Son, T. C. Adapting GOLOG for Composition of Semantic Web Services. *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning*, California, USA, 2002.

[79] Narayanan, S., and Mcilraith, S. A. Simulation, Verification and Automated Composition of Web Services. *Proceedings of the 11th international conference on World Wide Web*. New York, NY, USA, 2002.

[80] Mcdermott, D. V. Estimated-regression planning for interactions with web services. *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS'02)*, Toulouse, France, 2002.

[81] Waldinger, R. Web Agents Cooperating Deductively. *Proceedings of FAABS 2000*, Greenbelt, MD, USA, 2000.

[82] Lammermann, S. *Runtime Service Composition via Logic-Based Program Syndissertation*. PhD dissertation, Department of Microelectronics and Information Technology, Royal Institute of Technology, Sweden, June 2002.

[83] Rao, J., Kungas, P. and Matskin, M. Application of Linear Logic to Web Sservice Composition. *Proceedings of the 1st International Conference on Web Services*, Las Vegas, USA, June 2003.

[84] Rao, J., Kungas, P. and Matskin. Logic-based Web services composition: from service description to process model. *Proceedings of the 2nd International Conference on Web Services*, San Diego, USA, July 2004.

[85] Medjahed, B., Bouguettaya, A., and Elmagarmid, A. K. Composing Web Services on the Semantic Web. *VLDB Journal*, 12(4), 2003.

[86] Ponnekanti, S. R. and Fox, A. SWORD: A developer toolkit for Web service composition. *Proceedings of the 11th World Wide Web Conference*, Honolulu, USA, 2002.

[87] Sirin, E., Hendler, J., and Parsia, B. Semi-automatic Composition of Web Services Using Semantic Descriptions. *Proceedings of Web Services: Modeling, Architecture and Infrastructure Workshop*, Angers, France, 2002.

[88] Manna, Z. and Waldinger, R. Fundamentals of deductive program syndissertation. *IEEE Transactions on Software Engineering*, 18(8), 1992, 674-704.

[89] Wombacher, A., Fankhauser, P., and Neuhold, E. Transforming BPEL into Annotated Deterministic Finite State Automata for Service Discovery. *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, Washington, DC, USA, 2004.

[90] Kaynar, D.K., Lynch, N., Segala, R., and Vaandrager, F. The Theory of Timed I/O Automata Syndissertation, *Lectures in Computer Science*. Morgan & Claypool, 2006.

[91] Beek, M.H.T., Ellis, C.A., Kleijn, J., and Rozenberg, G. Synchronizations in Team Automata for Groupware Systems. *Computer Supported Cooperative Work*, 12(1), 2003, 21–69.

[92] Fu, X., Bultan, T. and Su, J. Analysis of Interacting BPEL Web Services. *Proceedings of the 13th international conference on World Wide Web*, New York, USA, 2004.

[93] Holzmann, G. J. The model checker SPIN. *Software Engineering*, 23(5), 1997, 279–295.

[94] Díaz, G., Pardo, J. J., Cambronero, M-E., Valero, V., and Cuartero, F. Automatic Translation of WS-CDL Choreographies to Timed Automata. *European Performance Engineering Workshop (EPEW 2005) and International Workshop on Web Services and Formal Methods (WS-FM 2005)*, Versailles, France, 2005.

[95] Larsen, K. G., Pettersson, P., and Yi, W. Uppaal: Status & Developments. *Proceedings of 9th International Conference on Computer Aided Verification*, Haifa, Israel, 1997.

[96] Dumas, M. , Wang, K. W. S., and Spork, M. L.. Adapt or Perish: Algebra and Visual

Notation for Service Interface Adaptation. *Proceeding of the 4th International Conference on Business Process Management (BPM'06)*, Vienna, Austria, Lecture Notes in Computer Science, Springer, 4102, 2006, 65–80.

[97] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes. *Journal of Information and Computing*, 100(1), 1992, 41–77, 1992.

[98] Salaün, G. Bordeaux, L., and Schaerf, M. Describing and Reasoning on Web Services Using Process Algebra. *Proceedings of IEEE International Conference of Web Services*, San Diego, USA, 2004.

[99] Milner, R. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice-Hall Inc., NJ, USA, 1989.

[100] Ferrara, A. Web Services: A Process Algebra Approach. *Proceedings of 2nd International Conference on Service Oriented Computing*, New York, USA, 2004.

[101] Bolognesi, T. and Brinksma, E. Introduction to the ISO Specification Language LOTOS. *Computer Networks*, 14, 1987, 25–59.

[102] Fernandez, J-C., Garavel, H., Kerbrat, A., Mounier, L., Mateescu, R., and Sighireanu, M. CADP - A Protocol Validation and Verification Toolbox. *Proceedings of 8th International Conference on Computer Aided Verification*, New Brunswick, USA, 1996.

[103] Reisig, W. and Rozenberg, G. Lectures on Petri Nets I: Basic Models, Advances in Petri Nets. *Lecture Notes in Computer Science*, Springer, 1491, 1998.

[104] Kiepuszewski, B., ter Hofstede, A.H.M., and van der Aalst, W.M.P. Fundamentals of Control Flow in Workflows. *Acta Informatica*, 39(3), 2003, 143–209.

[105] Ouyang, Chun, Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., and ter Hofstede, A.H.M. Formal Semantics and Analysis of Control Flow in WS-BPEL. *Scientific Computing Program*, 67(2-3), 2007, 162–198.

[106] Zhang, J., Chung, J-Y., Chang, C.K., and Kim, S. WS-net: A Petri-net based Specification Model for Web Services. *Proceedings of IEEE International Conference of Web Services*, San Diego, USA, 2004.

[107] Laliwala, Z., Khosla, R., Majumdar, P., and Chaudhary, S. Semantic and Rules Based Event-Driven Dynamic Web Services Composition for Automation of Business Processes. *IEEE Services Computing Workshops*, Chicago, USA, 2006.

[108] Cardoso, J. and Sheth, A. Semantic e-Workflow Composition. *Journal of Intelligent Information Systems*, 21(3), 2003.

[109] Cardoso, J. *Quality of Service and Semantic Composition of Workflows*. Ph.D. Dissertation. Department of Computer Science, University of Georgia, Athens, GA, 2002.

[110] Charkraborty, D., Perich, F. , Avancha, S., and Joshi, A. DReggie: A Smart Service Discovery Technique for E-Commerce Applications. *20th Symposium on Reliable Distributed Systems (SRDS)*. New Orleans, USA, 2001.

[111] Klein, M. and Bernstein, A. Searching for Services on the Semantic Web Using Process Ontologies. *International Semantic Web Working Symposium*, Stanford, USA, 2001.

[112] Srivastava, B. and Koehler, J. Web Service Composition – Current Solutions and Open Problem. *ICAPS 2003 Workshop on Planning for Web Services*, Trento, Italy, 2003 .

[113] Mao, Z. M., Brewer, E. R., and Katz, R. H. *Fault-tolerant, Scalable, Wide-Area Internet Service Composition*. January, 2001, U.C. Berkeley Technical Report UCB//CSD-01-1129.

[114] METEOR-S: Semantic Web Services and Processes, Available at lsdis.cs.uga.edu/proj/meteor/SWP.htm

[115] Patil, A, Oundhakar, S.A., Sheth, A., and Verma, K. METEOR-S Web Service Annotation Framework. *Proceedings of the 13th International Conference on World Wide Web (WWW ' 04)*, Manhattan, USA, 2004.

[116] Verma, K., Sivashanmugam, K., Sheth, A., Patil, A., Oundhakar, S.A., and Miller, J. METEOR–S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services. *Journal of Information Technology and Management*, 6(1), 2005.

[117] Sivashanmugam, K., Verma, K., Sheth, A., and Miller, J. Adding Semantics to Web Services Standards, *Proceedings of the International Conference on Web Services (ICWS ' 2003)*, Las Vegas, USA, 2003.

[118] Dietz, P. and Sleator, D. Two algorithms for maintaining order in a list. In *Proceedings of the 19th annual ACM Symposium on Theory of Computing (STOC),* New York, USA, 1987.

[119] Schenkel, R., Theobald, A., and Weikum, G. Efficient Creation and Incremental Maintenance of the HOPI Index for Complex XML Document Collections. *Proceedings of the 21st International Conference on Data Engineering* (*ICDE)*, Washington DC, USA, 2005.

[120] Wang, H., He, H., Yang, J., Yu, P.S., and Yu, J.X. Dual labeling: Answering Graph Reachability Queries in Constant Time. *Proceedings of the 22nd International Conference on Data Engineering*, Atlanta, USA, 2006.

[121] Roditty, L. and Zwick, U. A Fully Dynamic Reachability Algorithm for Directed Graphs With An Almost Linear Update Time. *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing (STOC ' 2004),* New York, USA, 2004.

[122] Trißl, S. and Leser, U. Fast and practical indexing and querying of very large graphs. *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, Beijing, China, 2007.

[123] Booth, D., et al. W3C Web Service Architecture Working Group. Available at http://www.w3.org/

[124] FIPA Agent Management Specification. Available at http://www.fipa.org/specs/fipa00023.

[125] Bellifemine, F., Caire, G., Greenwood, D. Developing Multi-agent Systems with JADE. John Wiley & Sons, West Sussex, England, 2004.

[126] Lange, D. B. and Chang, D. T. *IBM Aglets Workbench - Programming Mobile Agents in Java*. August 1996, White Paper, IBM Corporation, Japan.

[127] Searle, J. R. A Taxonomy of Illocutionary Acts. *Language, Mind, and Knowledge*, 7, 1975.

[128] Morelli, R. A., Goethe, J. W., and Bronzino, J. D.. A Language/Action Model of Human-Computer Communication in a Psychiatric Hospital. *Proceedings of the Annual Symposium on Computer Application in Medical Care*, Washington DC, USA, 1990.

[129] Greenwood, D. and Monique, C. Engineering Web Service-Agent Integration. *IEEE International Conference on Systems, Man and Cybernetics*, The Hague, Netherlands, 2004.

[130] Shafiq, M., Ding, Y., and Fensel, D. Bridging multi agent systems and web services: Towards interoperability between Software Agents and Semantic Web Services. *Proceedings of the 10th IEEE International Conference on Enterprise Distributed Object Computing* (*EDOC'06*), Hong Kong, China, 2006.

[131] Greenwood, D., Lyell, M., Mallya, A., and Suguri, H. The IEEE FIPA Approach to Integrating Software Agents and Web Services. *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems* (*AAMAS'07*), Honolulu, USA, 2007.

[132] Paolucci, M., Kawamura, T., Payne, T., and Sycara, K. Semantic Matching of Web Services Capabilities. *Proceedings of the First International Semantic Web Conference on the Semantic Web*. Sardinia, Italy, 2002.

[133] Sycara, K., Widoff, S., Klusch, M., and Lu, J. LARKS: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace. *Autonomous Agents and Multi-Agent Systems*, 5(2), 2002, 173–203.

[134] Klusch, M. Fries, B. and Sycara, K. OWL-S-MX: A hybrid Semantic Web Service matchmaker for OWL-S services. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(2), 2009, 121-133.

[135] Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., and Dean, M. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML* May 2004, Version 1.0, W3C member submission. Available at http://www.w3.org/Submission/SWRL/.

[136] Ait-Kaci, H., Boyer, R., Lincoln, P. and Nasr, R. Efficient implementation of lattice operations. *Programming Languages and Systems*, 11(1), 1989, 115–146.

[137] Caseau, Y. Efficient handling of multiple inheritance hierarchies. *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, New York, USA, 1993.

[138] Krall, A., Vitek, J. and Horspool, N. Near Optimal Hierarchical Encoding of Types. In *11th European Conference on Object Oriented Programming (ECOOP'97)*, Jyväskylä, Finland, 1997.

[139] Caseau, Y., Habib, M., Nourine, L. and Raynaud, O. Encoding of Multiple Inheritance Hierarchies and Partial Orders. *Computational Intelligence*, 15, 1999, 50–62.

[140] Bommel. M. and Beck, T. Incremental Encoding of Multiple Inheritance Hierarchies. *Proceedings of the Eighth International Conference on Information and Knowledge Management*, New York, USA, 1999.

[141] Agrawal, R., Borgida, A., and Jagadish, H. Efficient Management of Transitive Relationships in Large Data and Knowledge bases. *Proceedings of the ACM SIGMOD International Conference on Management of Data,* New York, USA, 1989.

[142] Constantinescu, I. and Faltings, B. Efficient Matchmaking and Directory Services. *Proceedings of the IEEE/WIC International Conference on Web Intelligence*, Beijing, China, 2003.

[143] Preuveneers, D. and Berbers, Y. Prime Numbers Considered Useful: Ontology Encoding for Efficient Subsumption Testing. 2006, Technical Report CW464, Katholieke Universiteit, Germany.

[144] Levesque, H. and Brachman, R. Expressiveness and Tractability in Knowledge Representation and Reasoning. *Computational Intelligence*, 3, 1987, 78-93.

[145] Mokhtar, S.B., Giorgantas, N., Issarny, V. COCOA : Conversation Based Service Composition for Pervasive Computing Environments. *Journal of Systems and Software*, 80 (12), 2007, 1941-1955.

[146] Bellur, U. and Kulkarni, R. Improved Matchmaking Algorithm for Semantic Web Services Based on Bipartite Graph Matching. *IEEE International Conference on Web Services* (ICWS 2007), Salt Lake City, USA, 2007.

[147] Klusch, M. Fries, B. and Sycara, K. Automated Semantic Web Service Discovery with OWLS-MX. *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems* (*AAMAS 2006*).

[148] Lara, R., Corella, M. A., and Castells, P. A Flexible Model for Web Service Discovery. *Proceedings of the 1st International Workshop on Semantic Matchmaking and Resource Retrieval* (*SMR 2006*), Seoul, Korea, 2006.

[149] Mokhtar, S.B., Preuveneers, D., Georgantas, N., Issarny, V., and Berbers, Y. EASY: Efficient semAntic Service discoverY in pervasive computing environments with QoS and context support. *Journal of Systems and Software*, 81(5), 2008, 785 – 808.

[150] Qiu, T., Li, L., and Lin, P. Web Service Discovery with UDDI Based on Semantic Similarity of Service Properties. *Proceedings of the 3rd International Conference on Semantics, Knowledge and Grid*, Xi'an, China, 2007.

[151] Lu, L.M., Chen, J.X., Zhu, G.J. Discovering Web Services Based on Semantic Web Technology. *International Conference on Next Generation Web Services Practices* (*NWeSP'05*), Seoul, Korea, 2005.

[152] Zhang, P. *The Research and Implementation of Semantic Based Web Services Discovery*. Dissertation of Master, Tsinghua University, China, 2005.

[153] Wang, G., Xu, D., Qi, Y., and Hou, D. A Semantic Match Algorithm for Web Services Based on Improved Semantic Distance, *4th International Conference on Next Generation Web Services Practices*, Seoul, S. Korea, 2008.

[154] Yang, H., Liu, S., Fu, P., Qin, H., and Gu, L. A Semantic Distance Measure for Matching Web Services. *Proceedings of the International Conference on Computational Intelligence and Software Engineering*, (*CiSE 2009*), Wuhan, China, 2009.

[155] Stoicay, I., Morrisz, R., Liben-Nowellz, D., Kargerz, D.R., Kaashoekz, M. F. et al. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1), 2003, 17-32.

[156] Crasso, M., Zunino, A., and Campo, M. AWSC: An Approach to Web Service Classification based on Machine Learning Techniques. *Inteligencia Aritificial*, 37, 2008, 25-36.

[157] Oldham, N., Thomas, C., Sheth, A., and Verma, K. METEOR-S Web Service Annotation Framework with Machine Learning Classification. *International Conference on Semantic Web Services and Web Process Composition*, San Diego, USA, 2004.

[158] Corella, M.A. and Castells, P. Taxonomy-Based Web service Categorization using Conceptual Parameter Descriptions. *Proceedings of the 1st Int'l Workshop on Semantic Matchmaking and Resource Retrieval: Issues and Perspectives (SMR 2006)*. Seoul, Korea, 2006.

[159] Corella, M.A. and Castells, P. Semi-Automatic Semantic based Web Service Classification. *Business Process Management Workshops*, Vienna, Austria, 2006.

[160] Duo, Z., Juan-Zi, L., and Bin, X. Web Service Annotation using Ontology Mapping. *IEEE International Workshop on Service-Oriented System Engineering*, (*SOSE 2005*), Beijing, China, 2005.

[161] Heß, A., Johnston, E., and Kushmerick, N. ASSAM: A Tool for Semi-Automatically Annotating Semantic Web Services. *Proceedings of the 3rd International Semantic Web Conference (ISWC 2004)*, Hiroshima, Japan, 2004.

[162] Lerman, K., Plangrasopchok, A., Knoblock, C.A. Automatically Labeling the Inputs and Outputs of Web Services. *Proceedings of 21st National Conference on Artificial Intelligence* (AAAI-06), Boston, USA, 2006.

[163] Heß, A. and Kushmerick, N. Learning to Attach Semantic Metadata to Web Services. *Proceedings of the 2nd International Semantic Web Conference (ISWC 2003)*, Sanibel Island, USA, 2003.

[164] Sajjanhar, A., Jingyu Hou, J., and Zhang, Y. Algorithm for Web Services Matching. *Proceedings of the 6th Asia-Pacific Web Conference (APWeb '04)*, Hangzhou, China, 2004.

[165] Wang, H., Shi, Y., Zhou, X., Zhou, Q., Shao, S., Bouguettaya, A. Web Service Classification using Support Vector Machine. *22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, Arras, France, 2010.

[166] Bruno, M., Canfora, G., Penta, M.D., and Scognamiglio, R. An Approach to Support Web Service Classification and Annotation. *Proceedings of the IEEE International*

Conference on e-Technology, e-Commerce and e-Service (*EEE'05*), Washington DC, USA, 2005.

[167] Saha, S., Murthy, C.A., Pal, S. K. Classification of Web Services Using Tensor Space Model and Rough Ensemble Classifier. *Proceedings of the 17th International Symposium on Methodologies for Intelligent Systems (ISMIS'08)*, Toronto, Canada, 2008.

[168] Liang., Q., Li, P., Hung, P.C.K., and Wu, X. Clustering Web Services for Automatic Categorization. *Proceedings of the 2009 IEEE International Conference on Services Computing (SCC '09)*, Bangalore, India, 2009.

[169] Shou, D. and Chi, C. Effective Web Service Retrieval Based on Clustering, *4th IEEE International Conference on Semantics, Knowledge and Grid*, Beijing, China, 2008.

[170] Corella, M. A. and Castells, P. A Heuristic Approach to Semantic Web Services Classification. *10th International Conference on Knowledge-Based & Intelligent Information & Engineering Systems* (*KES*). Universities of Brighton and Bournemouth, UK, 2006.

[171] Dong, X., Halevy, A., Madhavan, J., Nemes, E., and Zhang, J. Similarity Search for Web Services. *Proceedings of the 13th International Conference on Very Large Data Bases*. Toronto, Canada, 2004.

[172] Ma, J., Cao, J., Zhang, Y. A Probabilistic Semantic Approach for Discovering Web Services", *Proceedings of the 2007 IEEE World Wide Web Conference*, Banff, Canada, 2007.

[173] Ma, J., Cao, J., Zhang, Y. Efficiently Finding Web Services Using a Clustering Semantic Approach", *Proceedings of the 2008 International Workshop on Context Enabled Source and Service Selection, Integration and Adaptation (CESSS '08)*, Beijing, China, 2008.

[174] Gaber, J. and Bakhouya, M. An Affinity-driven Clustering Approach for Service Discovery and Composition for Pervasive Computing. *ACS/IEEE International Conference on Pervasive Services*, Lyon, France, 2006.

[175] Yuan-sheng, L., Yong, Q., Di, H., Ying, C., and Lin-feng, S. A Clustering and Selection Model for Service Composition using Granular Computing. *Proceedings of 4th IEEE Conference on Industrial Electronics and Applications (ICIEA 2009)*, Xian, China, 2009.

[176] Zhanga, L-J., Chengb, S., Cheea, Y-M, Allamc, A., Zhoua, Q. Pattern Recognition based Adaptive Categorization Technique and Solution for Services Selection. *Proceedings*

of the IEEE Asia-Pacific Services Computing Conference (APSCC '07), Tsukuba, Japan, 2007.

[177] Zamir, O., Etzioni, O., Madani, O., and Karp, R.M. Fast and Intuitive Clustering of Web Documents. *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, New Port Beach, USA, 1997.

[178] United Nations Standard Products and Service Code. Available at http://www.unspsc.org.

[179] The North American Industry Classification System. Available at http://www.census.gov/eos/www/naics/

[180] Bianchini, D., Antonellis, V., Pernici, B., and Plebani, P. Ontology-based Methodology for e-Service Discovery. *ACM Journal of Information Systems*, 31(4), 2006, 361 – 380.

[181] Platzer, C., Rosenberg, F., and Dustdar, S. Web Service Clustering using Multidimensional Angles as Proximity Measures. *ACM Transaction on Internet Technology*, 9(3), 2009, 54-79.

[182] Hau, J., Lee, W., and Darlington, J. A semantic similarity measure for semantic Web Services. *Proceedings of 2005 Web Service Semantics Workshop*. Chiba, Japan, 2005.

[183] Rada, R., Mili, H., Bicknell, E., and Blettner, M. Development and Application of a Metric on Semantic Nets. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(1), 1989, 17–30.

[184] Hirst, G., and St-Onge, D. Lexical chains as representations of context for the detection. *WordNet: An Electronic Lexical Database (Language, Speech, and Communication)*, The MIT Press, Cambridge, MA, 1995, 305–332.

[185] Resnik, P. Using information content to evaluate semantic similarity. *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, Montreal, Canada, 1995.

[186] Lin, D. An information-theoretic definition of similarity. *Proceedings of the 15th International Conference on Machine Learning*, Madison, USA, 1998.

[187] Keßler, C., Raubal, M., and Janowicz, K. The Effect of Context on Semantic Similarity Measurement. *On the Move to Meaningful Internet Systems: OTM 2007 Workshops*, Vilamoura, Portugal, 2007

[188] Borgida, A., Walsh, T., and Hirsh, H. Towards measuring similarity in description logics. *Proceedings of the 2005 International Workshop on Description Logics (DL2005)*, Edinburgh, Scotland, 2005.

[189] Dasgupta, S., Bhat, S., and Lee, Y. SGPS: a semantic scheme for web service similarity. *Proceedings of IEEE World Wide Web Conference*, Madrid, Spain, 2009.

[190] Godfrey, C., Siddons, A.W. *Modern Geometry (page 20)*. Cambridge University Press, London, UK, 1908.

[191] Sun Microsystems. *Jini Technology Core Platform Specification*. October 2000, Version 1.1. Available at http://www.sun.com/jini/specs.

[192] Perkins, C. Serice Location Protocol. *ACTS Mobile Networking Summit/MMITS Software Radio Workshop*, Rhodes, Greece, 1998.

[193] Zhao, W., Schulzrinne, H., Guttman, E. mSLP-Mesh-enhances Service Location Protocol. *Proceedings of the 9th International Conference on Computer Communications and Networks* (*ICCCN 2000*), Las Vegas, USA, 2000.

[194] Klyne, G., Reynolds, F., Woodrow, C., Ohto, H., Hjelm, J., Butler, M.H., Tran, L. *Composite Capability/Preference Profiles (CC/PP)*. January 2004, Version 1.0, W3C recommendation. Available at http://www.w3.org/TR/CCPP-struct-vocab/.

[195] Gioldasis N. and Christodoulakis S. Ubiquitous Web Applications. *Proceedings of E-business and e-work*, Prague, Czech Rupublic. 2002.

[196] Keller, A. and Ludwig, H. *The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Service.* May 2002, Technical Report RC22456(W0205-171), IBM Research Division, T.J. Watson Research Center.

[197] Tosic, V., Patel, K., and Pagurek, B. WSOL - Web Service Offerings Language. *International Workshop on Web Services, E-Business and the Semantic Web (WES 2002)*, Toronto, Canada, 2002.

[198] Lang, Q. and Su, S. AND/OR Graph and Search Algorithm for Discovering Composite Services. *International Journal of Web Services Research*, 2(4), 2005, 46-64.

[199] Nilsson, N. *Problem Solving Methods in Artificial Intelligence*. McGraw Hill, New York, 1971.

[200] Gu, Z., Xu, B., and Li, J. Service Data Correlation Modeling and Its Application in Data-Driven Service Composition. *IEEE Transactions On Services Computing*, 3(4), 2010.

[201] The Cyc Foundation. *OpenCyc Documentation*. Available at http://www.opencyc.org/doc.

[202] van der Aalst, W. M. P. and ter Hofstede, A. H. M. Yawl: Yet Another Workflow Language. *Elsevier Information Systems*, 30(4), 2005, 245–275.

[203] Strehl, A. and Ghosh, J. Value-based Customer Grouping from Large Retail Data-sets. *Proceedings of SPIE Conference on Data Mining and Knowledge Discovery*, Orlando, USA, 2000.

[204] Manning, C.D., Raghavan, P., and Schutze, H. *Introduction to Information Retrieval*. Cambridge University Press, USA, 2008.

[205] Lin, J. Divergence Measures based on the Shannon Entropy. *IEEE Transactions on Information Theory*, 37 (1), 1991, 145–151.

[206] Czerwinski, S.E., Zhao, B.Y., Hodes, T., Joseph, A.D., and Katz, R.H. An Architecture for a Secure Service Discovery Service. *Proceedings of the Fifth International Conference on Mobile Computing and Networks*, Seattle, USA, 1999.

[207] John, R. *UPnP, Jini and Salutaion—A Look at Some Popular Coordination Frameworks for Future Network Devices*. 1999, Technical Report, California Software Labs. http://www.cswl.com/whiteppr/tech/upnp.html.

[208] Mennie, D. and Pagurek, B. An Architecture to Support Dynamic Composition of Service Components. *Proceedings of the Fifth International Workshop on Component-Oriented Programming (WCOP)*, Sophia Antipolis, France, 2000.

[209] Chen, H., Joshi, A., and Finin, T. Dynamic Service Discovery for Mobile Computing: Intelligent Agents Meet Jini in the Aether. *Baltzer Science Journal on Cluster Computing* (*Special Issue on Advances in Distributed and Mobile Systems and Communication*), 2001, 343-354.

[210] Changyou, Z., Dongfeng, Z., Yu, Z., and Minghua. Y. A Web Service Discovery Mechanism Based on Immune Communication. *Proceedings of the International Conference on Convergence Information Technology (ICCIT '07)*, Gyeongju, Korea, 2007.

[211] Chakraborty, D., Joshi, A., Yesha, Y. Toward Distributed Service Discovery in Pervasive Computing Environments. IEEE Transactions on Mobile Computing, 5(2), 2006.

[212] Helal, S., Desai, N., and Lee, C. Konark-A Service Discovery and Delivery Protocol for Ad-Hoc Networks. *Proceedings of the Third IEEE Conference on Wireless Communication Networks* (*WCNC*), Mar. 2003.

[213] Undercoffer, J., Perich, F., Cedilnik, A., Kagal, L., Joshi, A., and Finin, T. A Secure Infrastructure for Service Discovery and Management in Pervasive Computing. *ACM MONET Journal (Special Issue on Mobility of Systems, Users, Data, and Computing*), 2002.

[214] Guttman, E., Perkins, C., and Veizades, J. *RFC 2165: Service Location Protocol*, June 1997.

[215] Ardissono, L., Goy, A., and Petrone, G. Enabling Conversations with Web Services. *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS '03)*, Melbourne, Australia, 2003.

[216] Benatallah, B., Casati, F., and Toumani, F. Web Service Conversation Modeling, A Cornerstone for E-Business Automation. *IEEE Internet Computing*, 8(1), 2004.

[217] Beringer, D., Kuno, H., and Lemon, M. *Using WSCL in a UDDI Registry 1.02*. 2001. Available at http://www.uddi.org/pubs/wsclBPforUDDI_5_16_011.doc.

[218] Willmott, S. and Dale, J. Agentcities: A Worldwide Open Agent Network. *Agentlink News*, 8, 2001, 13-15.

[219] Willmott, S., Dale, J , and Picault, J. The Agentcities Network Architecture. *Proceedings of the first International Workshop on Challenges in Open Agent Systems*, Bologna, Italy, 2002.

[220] Poggi, A., Tomaiuolo, M., and Turci, P. Service Composition in Open Agent Societies. *Workshop of Dagli Oggetti Agli Agenti (WOA)*, Villasimius, Italy, 2003.

[221] Poggi, A., Tomaiuolo, M., and Turci, P. An Agent-Based Service Oriented Architecture. *Workshop of Dagli Oggetti Agli Agenti (WOA)*, Genova, Italy, 2007.

[222] Poggi, A., Tomaiuolo, M., and Turci, P. Using Agent Platforms for Service Composition. *Proceedings of the 6th International Conference on Enterprise Information Systems* (*ICEIS*), Porto, Portugal, 2004.

[223] Toni, F. Argumentative KGP Agents for Service Composition. *AAAI Spring Symposium*, Stanford, USA, 2008.

[224] Toni, F. Assumption-based Argumentation for Selection and Composition of Services. *Proceedings of Computational Logic in Multi-Agent Systems (CLIMA-VIII)*. Porto, Portugal, 2007.

[225] CORDIS 6th Programme Framework. *ARGUGRID; Project Fact Sheet*. June 2002. Available at http://cordis.europa.eu/.

[226] Maamar, Z., Mostefaoui, S.K., and Yahyaoui, H. Toward an Agent-based and Context-oriented Approach for Web Services Composition. IEEE Transactions on Knowledge and Data Engineering, 17(5), 2005, 686 – 697.

[227] Buhler, P. and Vidal, J.M. Towards Adaptive Workflow Enactment Using Multiagent Systems. *Information Technology and Management Journal (Special Issue on Universal Enterprise Integration)*, 6, 2005, 61-87.

[228] Gelertner, D. and Carriero, N. Coordination Languages and their Significance. *Communications of the ACM*, 35(2), 1992, 97–107.

[229] McIlraith, S. A., Son, T. C. and Zeng, H. Semantic Web Services. *IEEE Intelligent Systems* (*Sp. Issue on The Semantic Web*), 16, 2002, 46–53.

[230] Sycara, K. P., Klusch, M., Widoff, S., and Lu, J. Dynamic Service Matchmaking Among Agents in Open Information Environments. *SIGMOD Record*, 28(1), 1999, 47-53.

[231] Papadopoulos, G. A. Models and Technologies for the Coordination of Internet Agents: A Survey. *Coordination for Internet Agents - Models, Technologies, and Applications*, Springer-Verlag, 2001.

[232] Buhler, P. and Vidal, J.M. Semantic Web services as Agent Behaviors. *Agentcities: Challenges in Open Agent Environments*, Springer, Berlin, 2003, 25–31.

[233] Wooldridge, M.J. *Reasoning About Rational Agents*. MIT Press, Cambridge, Massachusetts , USA, 2000.

[234] Bergenti, F. and Poggi, A. LEAP: A FIPA Platform for Handheld and Mobile Devices. *Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*, Seattle, USA, 2001.

[235] Achermann, F., Lumpe, M., Schneider, J-G. and Nierstrasz, O. Piccola - A Small Composition Language. *Formal Methods for Distributed Processing: A Survey of Object-Oriented Approaches*, Cambridge University Press, New York, USA, 2001, 403-426.

[236] Ketel, M. A Mobile Agent based Framework for Web Services. *Proceedings of the 47th Annual Southeast Regional Conference*. Clemson, USA, 2009.

[237] Ermolayev, V., Keberle, N., Plaksin, S., and Kononenko, O. Towards a framework for Agent-Enabled Semantic Web Service Composition. *International Journal of Web Services Research*, 1(3), 2004, 63-87.

[238] Salber, D., Dey, A.K., and Abowd, G.D. The Context Toolkit: Aiding the Development of Context-Enabled Applications", *Proceeding of the Conference on Human Factors in Computing Systems (CHI)*, Pittsburgh, USA, 1999.

[239] Chen, H., Finin, T., and Joshi, A. An Ontology for Context-aware Pervasive Computing Environments. *The Knowledge Engineering Review*. 18(3), 2003, 197-207.

[240] Chen, H., Finin, T., and Joshi, A. A Context Broker for Building Smart Meeting Rooms. *Proceedings of the Knowledge Representation and Ontology for Autonomous Systems Symposium* (*AAAI Spring Symposium*). Stanford, USA, 2004.

[241] Chen, H., Finin, T., and Joshi, A. The SOUPA Ontology for Pervasive Computing. *Ontologies for Agents: Theory and Experiences*, Birkhäuser, 2005, pp. 233-258.

[242] Dey, A. Understanding and Using Context. *Personal and Ubiquitous Computing Journal*, 5(1), 2001, 4-7.

[243] Endsley, M. R. Toward a Theory of Situation Awareness in Dynamic Systems. *Human Factors*, 37(1), 1995, 32-64.

[244] Toninelli, R., Bradshaw, J.M., Kagal, L., and Montanari, R. Rule-based and Ontology based Policies: Toward A Hybrid Approach to Control Agents in Pervasive Systems. *Proceedings of the Semantic Web and Policy Workshop*, Galway, Ireland, 2005.

VITA

Sourish Dasgupta was born on December 12th, 1980 in the city of Kolkata (formerly Calcutta) in India. He spent most of his childhood and schooling years in Durgapur – a small industrial city located near Kolkata. After coming out of high school he completed his Bachelors in Information Technology (BIT) from Manipal University, Karnataka, India in June 2003. He then continued to do his Masters in Computer Applications (MCA) from West Bengal University of Technology, West Bengal, India and graduated in June 2006. During his master's program Sourish developed a strong thirst for scientific research and therefore decided to come over to the United States of America for a PhD degree. He joined the I-PhD program at the Department of Computer Science Electrical Engineering in the University of Missouri – Kansas City, MO in fall 2006. His coordinating-discipline was Computer Science while his co-discipline was Telecommunication & Networking.

While pursuing his PhD Sourish started working under the supervision of Dr. Yugyung Lee (Associate Professor, Department of Computer Science Electrical Engineering). Sourish was initiated to some very interesting and challenging problems in the field of Service Oriented Architecture based intelligent distributed systems. Later on he got the opportunity to work on a National Science Foundation (NSF, USA) funded research project called ARTISAN (Art Inspired Service Oriented Architecture Design) under Dr. Yugyung Lee and Dr. Deendayal Dinakarpandian (Associate Professor, Department of Computer Science Electrical Engineering). In the course of his research Sourish has authored 9 peer-reviewed conference papers and two of his journal papers are under review. Sourish currently holds an assistant professor position at Dhirubhai Ambani Institute of

Information & Communication Technology (DA-IICT), Gandhinagar, India. His primary areas of research are theoretical and applied aspects of distributed multi-agent based intelligent cooperative models, service-oriented architecture based systems and semantic web modeling and mining. He is also highly interested in the field of computational cognition and information retrieval.

Publications

1. **Sourish Dasgupta**, Teja Swaroop Mylavarapu, Saurav Jana, Sudeep Maity, Yugyung Lee. "*SMARTSPACE: A Context-Aware Multi-Agent Platform for Distributed Service Discovery & Composition*". Technical Report L527FH_sd01V1.1, University of Missouri – Kansas City, April 2011.
2. **Sourish Dasgupta**, Yugyung Lee. "*Bit Encoding based Dynamic Polynomial Subsumption Testing of Description Logic Definitions*", Technical Report L527FH_sd02V1.1, University of Missouri – Kansas City, March 2011.
3. **Sourish Dasgupta**, Satish Bhat, Yugyung Lee. "*Event-driven Activity Logic Network for Service Oriented Systems*", IEEE Transactions on Services Computing, 2010 [under review]
4. **Sourish Dasgupta**, Satish Bhat, Yugyung Lee. "*Semantic Clustering for Web Service Discovery and Composition*", IEEE Transactions on Knowledge & Data Engineering. 2010 [under review]
5. **Sourish Dasgupta**, Satish Bhat, Yugyung Lee, "*Taxonomic Clustering and Query Matching for Efficient Service Discovery*", IEEE 9th International Conference on Web Services, Washington DC, USA, 2011

6. **Sourish Dasgupta**, Satish Bhat, Yugyung Lee, "*Taxonomic Clustering of Web Services for Efficient Discovery*", Proceedings of 19th ACM Conference on Information and Knowledge Management (CIKM), October 25 – 30, 2010
7. **Sourish Dasgupta,** Satish Bhat, Yugyung Lee, *CAOFES: An Ontological Framework for Web Service Retrieval*, Proceedings of 18th ACM Conference on Information and Knowledge Management (CIKM), November 2 – 6, 2009
8. **Sourish Dasgupta**, Satish Bhat, Yugyung Lee. "*An Abstraction Framework for Service Composition in Event-driven SOA systems*", IEEE 7th International Conference on Web Services, Los Angeles, CA, USA, 2009
9. **Sourish Dasgupta**, Satish Bhat, Yugyung Lee. *"Event Driven Service Composition for Pervasive Computing"*, IEEE International Conference on Pervasive Computing, Galveston, TX, USA, 2009
10. **Sourish Dasgupta**. "*A Logic-based Formalism for Pervasive Workflow*", IEEE International Conference on Pervasive Computing – PhD Forum, Galveston, TX, USA, 2009
11. **Sourish Dasgupta**, Satish Bhat, Yugyung Lee. "*Event Semantics for Service Composition in Pervasive Computing*", AAAI Spring Symposium, Stanford, CA, USA, 2009
12. **Sourish Dasgupta**, Satish Bhat, Yugyung Lee. "*SGPS: A Semantic Scheme for Context-Aware Event-driven Web Service Similarity*", 18th International World Wide Web Conference, Madrid, Spain, 2009

13. **Sourish Dasgupta**, Deendayal Dinakarpandian, Yugyung Lee. "*A Panoramic Approach to Integrated Evaluation of Ontologies in the Semantic Web*", 5th International EON Workshop, ISWC, Busan, S. Korea, 2007