PROVIDING SCALABILITY FOR AN AUTOMATED WEB SERVICE COMPOSITION FRAMEWORK

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ERTAY KAYA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JUNE 2010

Approval of the thesis:

PROVIDING SCALABILITY FOR AN AUTOMATED WEB SERVICE COMPOSITION FRAMEWORK

submitted by **ERTAY KAYA** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Adnan Yazıcı _____
Head of Department, **Computer Engineering**

Assoc. Prof. Dr. Nihan Kesim Çiçekli
Supervisor, **Computer Engineering Dept., METU** _____

Examining Committee Members:

Assoc. Prof. Dr. Ferda Nur Alpaslan
Computer Engineering Dept., METU _____

Assoc. Prof. Dr. Nihan Kesim Çiçekli
Computer Engineering Dept., METU _____

Assoc. Prof. Dr. Halit Oğuztüzün
Computer Engineering Dept., METU _____

Assist. Prof. Aysu Betin Can
Informatics Institute, METU _____

Dr. Ayşenur Birtürk
Computer Engineering Dept., METU _____

Date: 07.06.2010

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Ertay KAYA

Signature :

ABSTRACT

PROVIDING SCALABILITY FOR AN AUTOMATED WEB SERVICE COMPOSITION FRAMEWORK

Kaya, Ertay

M.Sc., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Nihan Kesim Çiçekli

June 2010, 104 pages

In this thesis, some enhancements to an existing automatic web service composition and execution system are described which provide a practical significance to the existing framework with scalability, i.e. the ability to operate on large service sets in reasonable time. In addition, the service storage mechanism utilized in the enhanced system presents an effective method to maintain large service sets. The described enhanced system provides scalability by implementing a pre-processing phase that extracts service chains and problem initial and goal state dependencies from service descriptions. The service storage mechanism is used to store this extracted information and descriptions of available services. The extracted information is used in a forward chaining algorithm which selects the potentially useful services for a given composition problem and eliminates the irrelevant ones according to the given problem initial and goal states. Only the selected services are used during the AI planning and execution phases which generate the composition and execute the services respectively.

Keywords: Automatic web service composition and execution, semantic web services, AI planning, scalability, service filtering.

ÖZ

BİR OTOMATİK WEB SERVİS BİLEŞİM ÇATISINA ÖLÇEKLENİRLİK SAĞLAMA

Kaya, Ertay
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Nihan Kesim Çiçekli

Haziran 2010, 104 sayfa

Bu tezde, var olan bir otomatik örün servis bileşim ve çalıştırma sistemine yapılan bazı iyileştirmeler tarif edilmektedir. Bu iyileştirmeler, var olan sisteme ölçeklenirlik (büyük servis kümeleri üzerinde makul zamanda çalışma yeteneği) sağlayarak uygulama açısından önem kazandırmaktadır. Ayrıca, iyileştirilmiş sistemde kullanılan servis depolama düzeneği, büyük servis kümelerini muhafaza etmek için etkili bir yöntem sağlamaktadır. Tarif edilen iyileştirilmiş sistemde ölçeklenirlik, servis zincirlerini ve servislerin verilen problemdeki başlangıç ve hedef durumlarına bağımlılıklarını bulan bir ön işleme safhasıyla sağlanmaktadır. İyileştirilmiş sistemdeki servis depolama düzeneği, bu bulunan bilgileri ve var olan servislerin tanımlarını depolamak için kullanılmaktadır. Bulunan bilgiler, verilen bir bileşim problemi için, problemin başlangıç ve hedef durumlarına gore potansiyel olarak kullanılabilir olan servisleri seçen ve ilgisiz olanları eleyen bir ileri yöne zincirleme algoritmasında kullanılmaktadır. Sırasıyla servis bileşimini bulan ve çalıştıran, yapay zeka planlama ve çalıştırma safhalarında sadece bu seçilmiş servisler kullanılmaktadır.

Anahtar kelimeler: Otomatik örün servis bileşimi ve çalıştırması, anlamsal örün servisler, yapay zeka planlama, ölçeklenirlik, servis filtreleme.

To my family…

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and appreciation to my supervisor Assoc. Prof. Dr. Nihan Kesim Çiçekli for her any kind of support and encouragement throughout this study. It was a complete privilege working with such a friendly, considerate and intellectual supervisor.

I am deeply grateful to my dear friend Mehmet Kuzu for his invaluable support to this work with his extensive knowledge and ideas. His contribution helped me to overcome many difficulties easily.

I would also like to convey thanks to jury members for their valuable comments on this thesis.

I would like to thank to my family for their never ending love and concern.

I am indebted to my precious friends Duygu Ceylan, Seda Çakıroğlu, Goncagül Demirdizen, Hilal Karaman, Eda Kılıç, Cenk Özkan, Gizem Öztürk and Anıl Sınacı for their encouragement and sharing experiences about their thesis studies.

As a final acknowledgement, I would like to thank to Peter Bartalos from Slovak University of Technology in Bratislava for answering my questions with patience and clarifying all the obscure points in my mind about his web service composition system.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

With the improvements in web technologies and the growing scale of software systems, the interoperability among the distributed software modules has become a crucial concern. Interoperable distributed systems enable to benefit from ready-to-use software modules and this provides cost and time effective software development. To achieve interoperability, loose coupling among the software modules should be assured. Web services are the most widely used way of implementing distributed systems as they provide the required loose coupling among distinct systems and interoperable distributed software environments. Web services provide ready-to-use functionalities through fixed interfaces for other applications by hiding the implementation details and they are commonly used in real world applications.

With the increasing number of available web services, maintaining these services and searching for the ones that satisfy a given requirement has become an important problem. Moreover, in the cases where a single service cannot satisfy a requirement, a composition of more than one service should be created and run to fulfil this requirement. In the environments including many web services, as these services cannot be searched manually, automation of the discovery and composition processes is mandatory. The improvements in the semantic web and semantic web service technologies help to automate these processes. Syntactic descriptions of web services are done with WSDL [2] which presents the required physical execution information such as service endpoint, network communication protocol and syntactic service input and output message definitions. These descriptions are not sufficient to

find a service that fulfils a requirement or to create a composition from more than one service. Semantic descriptions of services are required at this point which enable the machine interpretation of the service data. These semantic descriptions are done with OWL [3] and OWL-S [4] semantic web languages.

In addition to using semantic descriptions of web services for discovery and composition, the similarity between web service composition and AI planning problems enable the usage of AI planners in web service composition systems. The semantic descriptions of web services can easily be converted to a planning domain with some defined rules and this domain can be used to find a plan for a given composition request.

In the literature, there exist many approaches that attempt to find solutions to the web service composition problem. [8], [11] and [26] discuss some of these approaches with their drawbacks and benefits. [10] and [11] identify the open problems and desirable aspects of web service composition systems. [1] describes an existing dynamic web service composition and execution (WSCE) framework which provides a comprehensive solution to many of these problems and desirable aspects. This framework interleaves web service composition and execution with the help of an AI planner and this approach provides resiliency to dynamic execution environments. For example, in case a service fails to execute, the framework revises the created composition and tries to find other solutions for the problem. In addition, in the scenarios that web services have some nondeterministic effects, they can be observed after the real execution and state change is done by the planner [1]. Although this framework successfully addresses many open problems, it has two important deficiencies, namely scalability and service data maintenance, which decrease its practical significance and prevents it from being applicable for real world environments and scenarios.

Firstly, the existing WSCE framework fails to be scalable, i.e. to be able to operate on large service sets in reasonable time. Scalability is a critical issue for a web service composition (WSC) system because it is quite common that real world

service domains include thousands of services to be composed and the WSC system cannot achieve practical significance without being able to deal with such service sets. The reason that prevents the existing WSCE framework from being scalable is the fact that it utilizes a domain-independent AI planner for the composition generation. Most domain independent planners fail to respond in reasonable time if the search space becomes very large. Furthermore, AI planners are generally built for classical planning problems. These problems mostly include small number of actions which have large number of preconditions and effects. WSC problems do not fit well into classical planning problems because they include a large number of services (i.e. actions) which have small number of preconditions and effects.

Using domain-dependent planners may be considered as a solution to the scalability problem. [26] compares different planning approaches for WSC problem and shows that domain-dependent planners scale well with problems that have large domains. However, as the domain of WSC problems cannot be specified when services in the whole web are considered, using domain-dependent planners prevents the applicability of WSC systems to real world problems.

Since the domain-independent planners fail to be scalable and domain-dependent planners are not suitable for WSC problems, a promising solution emerges as applying some extensions to domain-independent planners to provide scalability. The most widely used mechanism for WSC systems is to implement some pre-filtering on the service domain before invoking the planner to find the composition. With the help of pre-filtering, the planner runs on a smaller set of candidate actions and returns the plan quickly.

Pre-filtering process makes use of the input-output and precondition-effect matching of services with each other and with the initial and goal states of the composition problem. Thus, it is dependent on the initial and goal states of the composition problem and begins after the user provides this information. As a result, the time required for this process should also be kept as small as possible to enable a timely response to user requests. A solution to this problem is to preprocess the service

domain and exploit the composition independent information that will help to decrease the time required for pre-filtering as much as possible. An example of problem independent information that can be found with preprocessing is the input-output and precondition-effect matches between two services. As these matches are independent of the given composition problem, the service domain can be analyzed and such matches can be found and recorded before trying to solve a given problem.

As the purpose of the preprocessing is to exploit and record some additional information about the service domain, the storage medium, information access-update efficiency and space complexity become important issues that need to be handled effectively. A well designed relational database (RDB) can help to keep space complexity small. In addition RDBs provide a simple and clean interface for information access-update with the help of the power of SQL. The problem of RDBs is that they are located on disk which has a negative effect on the information access-update performance. This directly affects the performance of the pre-filtering process since it accesses the information very frequently while finding the candidate services for the composition. Using an in-memory database (IMDB) [27] instead of a disk database increases the performance of the system significantly as it enables in-memory access and update of the information.

The second problem of the existing WSCE framework is its inability to maintain service domains. Since the number of available web services is generally huge in real world domains, a WSC system should also be able to maintain these huge service sets in a service repository and select the candidate services from this repository depending on some search criteria during composition. In [1], when the user wants to run the framework for a composition problem, he/she needs to provide the WSDL and OWL-S descriptions of all web services as well as the OWL description of the problem to be solved. This approach brings some problems in terms of system performance and ease-of-use.

Firstly, since the existing WSCE framework uses an AI planner that uses PDDL domain and problem descriptions, a language conversion is required from

OWL/OWL-S to PDDL. Since the framework needs OWL/OWL-S description of services for each problem, the conversion from OWL/OWL-S to PDDL takes place each time the framework is invoked. This situation leads to unnecessary processing because same OWL/OWL-S descriptions may be converted to PDDL for many times. In addition, the conversion process in [1] is triggered after the user provides the composition problem. This increases the time required to find a solution to the problem. Maintaining a service repository can be a solution to these problems. If the PDDL descriptions of web services are stored in this repository, no repeated conversion will be done for a service. Furthermore, these PDDL descriptions can directly be picked from the repository and used for the composition when a problem is given, without requiring an additional time for the language conversion.

Secondly, the existing WSCE framework expects the semantic service descriptions of all available services from the user as a single OWL-S file. The same is also valid for syntactic service descriptions in which case the framework expects a single WSDL file from the user. This is not a practically applicable solution because this approach provides a very weak service domain maintenance mechanism when huge numbers of services are considered in real world scenarios. When a service becomes unavailable, it is quite complex and time consuming to find and remove the semantic and syntactic descriptions of the service. The same problem also occurs when a new service is wanted to be added to the service domain. This approach also decreases the usability of the framework because the data that a user friendly WSC system asks from the user should only be a problem description. Service descriptions should be collected from the service providers and this collection process should be independent of any composition request. Furthermore, in most cases the user may not be able to access the whole set of services from different publishers and provide this set to the framework to solve a composition problem. All these problems can be handled by adding a service repository to the existing WSCE framework. A service repository provides an internal storage system for services so that services can easily be stored and updated anytime independent of any composition request. In addition,

the user does not need to provide any service description with the composition problem.

This thesis aims to enhance the existing WSCE framework by providing practical significance with service domain maintenance and scalability. The contributions of this thesis can be briefly described as follows:

- This thesis supplements the existing WSCE framework with a service repository. The repository stores the syntactic and semantic information about the services. The stored service information is used while dealing with the composition problem to find the services to be involved in the composition. In addition to the service data, the service repository stores some additional information about the services, namely chaining information between services and problem state dependencies. This additional information helps to decrease the time required for pre-filtering step of the composition.
- The existing WSCE framework is also enhanced with a pre-processing mechanism. This mechanism processes the service information retrieved from the service providers and stores the required information in the service repository. In addition, this mechanism extracts the service chains and the problem state dependencies and stores this information as well. With the help of this pre-processing mechanism, service information can be stored in the service repository without requiring the user to provide it together with the composition problem.

- The enhanced system in this thesis runs a pre-filtering process before invoking the AI planner in [1]. This process quickly finds the reduced service set that will be forwarded to the AI planner to find the composition. This reduced set contains only the services that have the possibility to be involved in the composition for the given problem. This reduction provides scalability

to the system and it is achieved by using the chaining and problem state dependency information stored by the pre-processing mechanism.

- When a service is found to be unreachable during the service execution phase in [1], the enhanced system removes the information related to this service from service repository.

The organization of the thesis is as follows: Chapter 2 presents some background information about the concepts and technologies used in this thesis. In addition, some important related work is explained in Chapter 2 and they are compared with this thesis in terms of their weaknesses and strengths. Chapter 3 explains how the service information is preprocessed and stored in the service repository. In Chapter 4, the details of the pre-filtering mechanism are described and it is explained how it provides scalability to the system. The integration of this pre-filtering mechanism with the preprocessing phase of the existing WSCE framework is explained in Chapter 5. Chapter 6 gives the experimental results that show the performance of the pre-filtering mechanism and Chapter 7 concludes the thesis with a brief summary and some future work.

CHAPTER 2

BACKGROUND INFORMATION AND RELATED WORK

This chapter consists of two parts. In the first part, some background information regarding the terminology, standards and systems used in this thesis is presented. In addition, definitions and general overview of the web service discovery and composition concepts are explained in this section.

In the second part, some approaches that utilize service filtering algorithms to eliminate redundant services are described and their comparison with this thesis work is made. Furthermore, the existing WSCE framework is described with its contributions to the WSC literature and its deficiencies.

2.1 Background Information

2.1.1 Semantic Web Services and OWL-S

In today's software systems, achieving interoperability between distributed and distinct entities which use different standards and platforms is quite important. Service Oriented Architectures (SOAs) and their core components, web services, are the widely used approaches to provide the required interoperability.

The term "web service" is formally defined by W3C as "a software system designed to support interoperable machine-to-machine interaction over a network." [14]. Web services are created by service providers and consumed by client agents by complying with some predefined rules. Web services allow the service providers to create and publish a functional interface for the service consumers without

disclosing the implementation details of the provided functionality. This type of communication assures loose coupling between different applications which means clients can access and use the provided services independent of the hardware platform, operating system, programming language etc.

SOAs conform to some standards in order to be able to assure the interoperability between different entities. The most important standards are Web Service Description Language (WSDL) [2] for service description, Simple Object Access Protocol (SOAP) [15] for messaging format and Universal Description Discovery and Integration (UDDI) [16] for publishing the services to clients.

The advantages of web services mentioned above allowed them to be used commonly in the recent years. On the other hand, the increase in the number of available services resulted in some other expectations from the real world that are waiting to be satisfied. [13] lists these expectations as follows:

Automatic discovery of a web service: Finding a desired service can be difficult, especially if the client does not know the existence of the service requested. Some mechanisms should be provided to allow automatic discovery of the existing services.

Automatic invocation of a web service: After finding the requested service, the software agent should be able to invoke the service automatically. This is especially important for providing efficiency to large-scale businesses.

Automatic composition of necessary web services: It is quite common that a specific goal requires several web services to work together. A software agent should be able to find all the necessary services and create a composition that achieves the given goal.

Automatic monitoring of the execution process: If the preceding processes become automatic, some mechanisms will be required to detect and report possible failures.

Unfortunately, the standards mentioned above are not sufficient to achieve these goals. SOAP is mainly for low-level data exchange and used only during service execution. However, it does not help to the automation of service invocation. WSDL descriptions are insufficient because they do not include any semantics about the service. They only describe the interface and this interface can be common for many services achieving different functionalities. Since UDDI entries also point to the WSDL documents of the services, they also lack semantic annotation.

The required extensions to these standards are provided with the Semantic Web Services (SWSs). The term semantic web service can simply be defined as "a web service with explicit semantic annotation" [13]. The goals of automating service discovery, invocation, composition and monitoring can be achieved with the combination of semantic annotations provided by the SWSs and the standards mentioned above.

Two different paths have been proposed for the SWSs. One of these paths is to add semantic annotations to the current standards such as UDDI and WSDL. The main advantage of this path is the ability to reuse these widely accepted standards. The other path is to create stand-alone semantic descriptions and upper ontologies based on some universally agreed ontologies. With the help of such descriptions, an automatic agent will have sufficient information for discovery, invocation, composition and monitoring [13]. The most widely accepted upper ontology is OWL-S [4].

OWL-S:

OWL-S, which stands for "Web Ontology Language - Services" is currently the standard for web service annotation. It is written using OWL and its goal is to provide general terms and properties to describe web services [13]. This standard consists of three parts: Service profile, service model and service grounding [4].

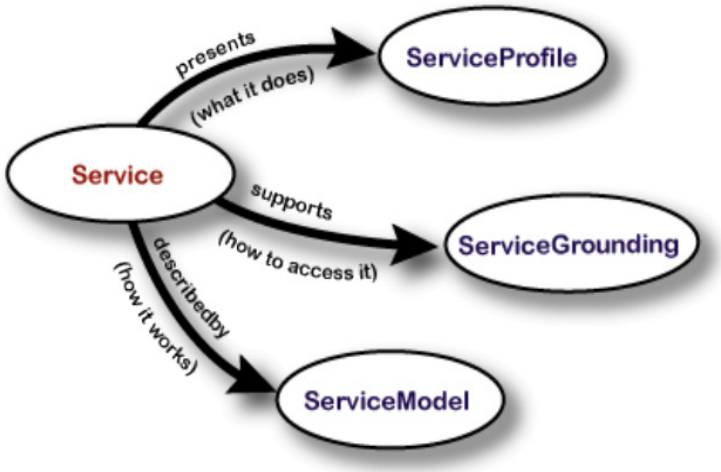

Figure 2.1: OWL-S model

Figure 2.1 which is adapted from [4] illustrates the OWL-S components and relationships among them. According to OWL-S specification, these components are used as follows [4]:

ServiceProfile: The profile of a service explains "what the service does" in a suitable way for a service-seeking agent so that the agent can determine whether the corresponding service meets its requirements. The ServiceProfile representation covers a description of what is accomplished by the service, the limitations of the service applicability, quality of service (QoS), and the preconditions that the agent must satisfy to invoke the service successfully.

ServiceModel: The model of a service explains "how an agent uses the service". This is done by detailing the semantic content of requests, the conditions under which some particular outcomes will occur, and the step by step processes leading to those outcomes, where necessary. In other words, ServiceModel describes how to ask for the service and what happens when the service is executed.

ServiceGrounding: The grounding of a service explains the details of how an agent can access the service. Typically it specifies the communication protocol, message

formats and other service-specific details like port numbers used for contacting the service. Furthermore, the grounding specifies an unambiguous way of exchanging data elements of a type of input or output specified in the ServiceModel.

2.1.2 AI Planning and PDDL

Planning is defined in [19] as "the task of coming up with a sequence of actions that will achieve a goal". A search based, propositional or first order logic based software agent can also be regarded as an AI planning agent, but they are very primitive and cannot be used in large domains and real world applications. For example, the invocation of a simple service that provides the path between two different locations is infeasible for a particular problem in the case that the total number of locations is high because the search space is proportional to the square of the number of available locations. If there are 1000 locations, 1 million locations are contained in the search space. Thus some heuristics are required for planning problems as well as some useful domain knowledge to extract these heuristics.

AI planners use some sort of planning languages such as PDDL [5] to define the domain and the problem itself. The representational power of these languages enables to extract some heuristics that help to prune the search space. A planning problem is generally described with state and action combinations. States are conjunctions of positive and negative literals that describe the world and actions are the operators that cause the state changes. Actions consist of preconditions and effects. In order to execute an action, the current state should satisfy the preconditions of the action. After the execution of the action, the current state changes to a new state that includes the effects of the action.

There exist various paradigms for planning. The primitive ones are based on state space search algorithms like forward chaining and backward chaining [19]. In forward chaining, the direction of the search is from the initial state to the goal state and vice versa for backward chaining. Generally speaking, backward chaining is a better approach because it provides lower branching factor. Both approaches find

total order plans and do not use effective heuristics. Therefore, they are not applicable for most of the real world problems because of their high computational complexities. Partial order planners (POPs) use a more efficient approach. Instead of considering the problem as a whole and finding total plans, POP divides the problem into smaller parts which has a contribution to decreasing the computational complexity [19]. A better approach is Graphplan which uses a data structure called planning graph and benefits from the important heuristics like mutual exclusion (mutex) relations among literals and actions which are extracted from the planning graph [20]. The mentioned approaches are the basic underlying planning algorithms of various planners that are used in the real world domains. The real world planners generally consider time constraints, nondeterminism, partial observability and scalability issues and make important additions to these basic algorithms. For example, HTN planners [19] are very similar to POPs but they use task decomposition which provides scalability by reducing the time complexity, but they need some additional domain information to achieve this.

PDDL:

The Planning Domain Definition Language (PDDL) [5] is the de-facto standard as the input language for most of the modern AI planners [19]. It has sublanguages for STRIPS [21], ADL [22] and HTN domains. PDDL definitions consist of two different parts: domain and problem definitions. The domain definition contains available actions, predicates and types. The problem definition contains the initial state, goal state and available objects. Figure 2.2 shows the format of PDDL domain description as stated in [23].

```
(define (domain <domain name>)
        <PDDL code for predicates>
        <PDDL code for first action>
        [...]
        <PDDL code for last action>
)
```

Figure 2.2: PDDL Domain File Format

Predicates represent the object and data type properties that exist between objects. Actions represent the semantic meaning of the operations and provide information about the inputs and outputs of the operations as well as precondition and effect specifications. Figure 2.3 shows the format of PDDL problem description.

```
(define (problem <problem name>)
    (:domain <domain name>)
    <PDDL code for objects>
     <PDDL code for initial state>
     <PDDL code for goal specification>
)
```

Figure 2.3: PDDL Problem File Format

In this figure, objects represent the available physical and conceptual components. Initial state represents the current state of the available objects and the goal state represents the desired state of the available objects.

2.1.3 Web Service Discovery

With the rapid increase in the number of available web services in the past few years, finding suitable web services that provide a requested operation has become an important problem for service oriented systems. This problem has been addressed by many web service discovery approaches. Web Service Discovery is broadly

described as “the act of locating a machine-processable description of a web service-related resource that may have been previously unknown and that meets certain functional criteria” [14]. This section explains the role of service discovery in the web service model and summarizes the types of web service discovery systems with their benefits and drawbacks.

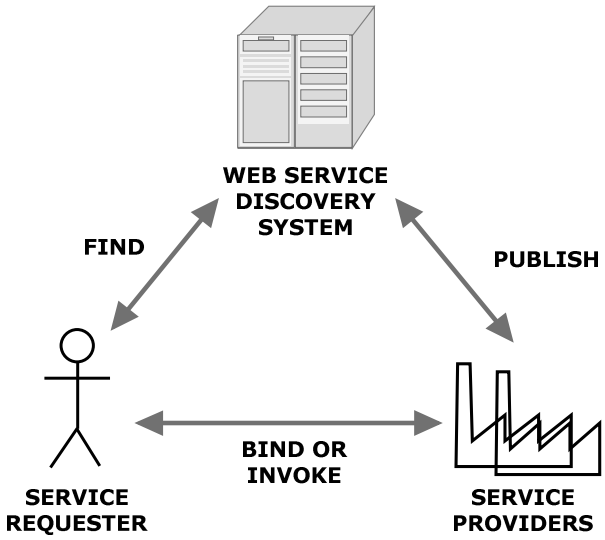

Figure 2.4: Web service invocation model

Figure 2.4 which is adapted from [18] illustrates the web service invocation model. In this figure, service providers are the entities that offer web services. These services are described with Web Service Description Language (WSDL) [2]. The service requester describes the required service to find the suitable providers. The web service discovery system is a broker that provides the service registry and search functions. The providers advertise the available services in this system. The system is searched for the requests retrieved from the requester. As the registry standard, Universal Description Discovery and Integration (UDDI) [16] is used in the discovery system. The operations in Figure 2-4 are briefly described as follows [18]:

Service Requester: The requester describes the required service

Publish: Service providers publish the available services for requesters via a discovery system. With this operation, the description of the service available at provider side is stored in the discovery system.

Discovery: The service requester retrieves the service provider information from the discovery system. The discovery system uses the description of the requested service provided by the requester and returns a list of suitable service providers.

Bind: After retrieving the list of service providers, the requester invokes these providers and creates a binding at runtime. The requester and providers use Simple Object Access Protocol (SOAP) [15] which is an XML-based protocol used for web service-client communication.

The problems in the current web service discovery approaches are explained in[24]. It is stated that the main obstacle that affects the web service discovery mechanisms is the heterogeneity among the services. The heterogeneities are classified as technological heterogeneities which are caused by different platforms or data formats, pragmatic heterogeneities that result from different development of domain-specific processes, and ontological heterogeneities which are caused by domain-specific terms and concepts, the description language used to describe the ontologies and the coexistence of both semantic and non-semantic web services.

The web service discovery problem is divided into two categories as semantic web service discovery and non-semantic web service discovery as illustrated in Figure 2.5 which is adopted from [18]. The semantic web service discovery is also divided into two sub-categories which are explained as follows:

Discovery of semantic web services using the same ontology: This type of discovery is used by most of the systems proposed. Some of these systems match the request and service profiles directly, some divide the matching into several stages and others make use of UDDI for matching. In direct matching systems, users are

not permitted to interfere with the matching process. These systems are accurate but time consuming. In order to allow the user interference and time efficiency, some systems divide the matching process into several stages. In these approaches, the degree of the similarity between the request and the service can be specified and constraints can be defined to enhance the result. The approaches that use UDDI for matching make some enhancements to the standard by including semantic web service profile because UDDI's registry mechanism lacks semantics [18].

Discovery of semantic web services using different ontologies: This type of discovery covers the cases where the web service requester and the provider use different ontologies. As the web service requesters and providers operate independently in the real world, each defines their own ontologies to describe their services. A service provider can be providing a service required by a requester although they use different ontologies. It is an open research topic to implement discovery systems which find matches in different ontologies [18].

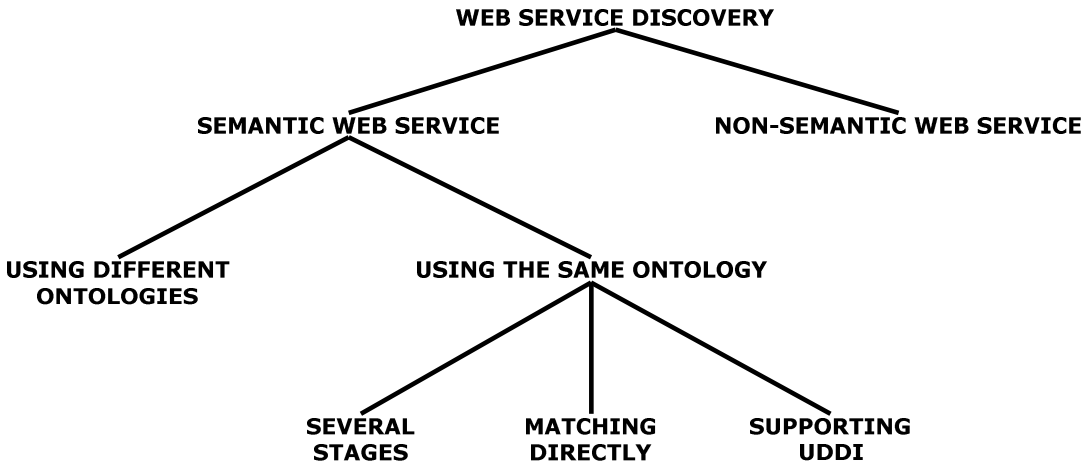

Figure 2.5: Web service discovery categories

2.1.4 Web Service Composition

There exists a huge number of web services that can handle particular requests. When a complex request cannot be handled by a single service, a combination of more than one service is required to handle this request. This process of combining web services to achieve complex tasks is called Web Service Composition (WSC). The resultant service is called a Composite Service which is defined as "a set of atomic services together with the control and data flow among the services" [12]. WSC is a hot research topic and there exists a considerable amount of work in this area.

Figure 2.6 which is adapted from [12] illustrates the overall architecture of WSC frameworks. Most of the approaches proposed in this area conform to this abstract framework. The steps that are followed in this framework are as follows [12]:

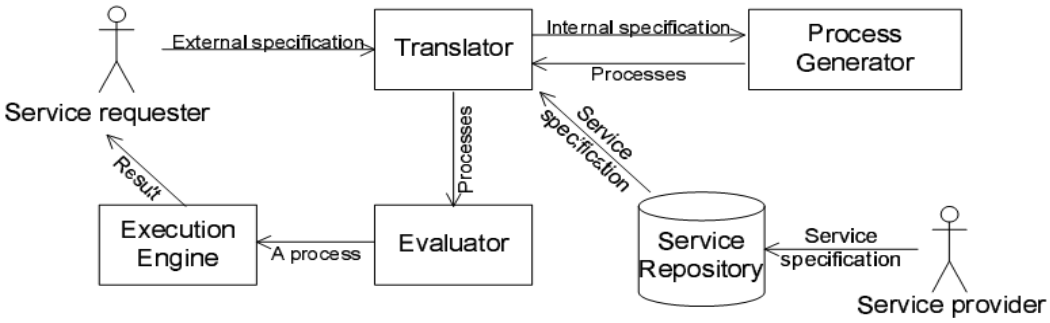

Figure 2.6: Web service composition framework

Presentation of a single service: In this phase, the signature of the service is presented by the service providers. This signature includes the semantic descriptions like preconditions, world altering and information providing effects are provided as well as the input and output parameters. In addition, some non-functional properties

are also described such as transactional and quality of service (QoS) attributes. This service information is generally contained in UDDI registries.

Translation of the languages: Generally, the languages that are used for problem descriptions and semantic web service annotations are not directly interpretable by workflow engines or AI planners which provide the solution. Thus, a translation between languages is needed for solution providing engines to understand the problem and the available service domain.

Generation of the composition process model: This phase is the most important phase in the WSC framework where the AI planner or the workflow engine takes place. This phase provides a solution to the given WSC problem. The main difference between many approaches comes from the systems used in this phase.

Evaluation of the composite service: In some cases more than one possible solution is provided by the previous phase. In this case an evaluation of the available solutions is done to decide the best solution. This phase can be completed with human intervention or software agents which make use of some heuristics.

Execution of the composite service: After a solution is found for the given problem, the services can be invoked in the execution environment. After the services are determined, the generated client stubs invoke the services via some RPC calls.

There are two approaches commonly accepted for the WSC problem which are using workflow engines and using AI techniques [12]. Although there are some approaches that use workflow engines such as EFlow [25], they are not as successful as the approaches using AI methodologies because these approaches require a considerable amount of human intervention to find a composite service for the given problem and they provide limited flexibility. For example, EFlow generates the abstract service composition manually; the only automated process is to bind these abstract service definitions to concrete services [25].

Different AI planning techniques have been proposed in many approaches to enable automated WSC. The adequacy of the used planners is vital for the correctness of the composite service. [26] discusses different approaches that use different AI planning methodologies and compares these approaches in terms of different criteria. Table 2.1 which is adapted from [26] shows the comparison of these approaches in terms of domain dependence, domain complexity and scalability. The reasons for choosing of these criteria are illustrated in Table 2.2 which is also adapted from [26].

Table 2.1: Comparison of AI planning techniques applied to WSC

Criterion	**Domain-independent Heuristic (Optop)**	**HTN Planning (SHOP2)**	**Situation Calculus (Golog)**	**Planning as model checking (ASTRO)**	**Planning based on Markov Decision processes**
Domain-independence	Domain-independent	Domain-specific or domain-configurable	Domain-independent	Domain-specific	Domain-specific
Domain complexity	Building the regression-match graph is too expensive (no proper measurement exists)	$O((n/k)!\cdot k)$	Plan generation takes linear time. Combination of complex actions is polynomial in the number of primitive action occurrences in its definition	$O(n!)$	$O(\|S\|^2)$
Scalability	Not feasible	Scales well to large domain problems	Reasonable	Can deal with large scale problems	Lack of scalability

Table 2.2: Comparision Criteria

Criterion	**Reason of choice**
Domain-independence	Allows the solution of a broad range of problems.
Domain complexity	Measures the level of difficulty in solving composition problem in terms of time and computation steps.
Scalability	Provides the ability to solve large real world problems.

As it can be inferred from Table 2.1, it is still not possible to provide good scalability by the systems using domain-independent planners because these planners fail to respond if the search space becomes very large. The systems using situation calculus show reasonable scalability but this is also not sufficient for real world problems which may contain thousands of services in their domains. On the other hand, some of the approaches using domain-dependent planners scale well with the growing domain size. The problem in these approaches is that they are able to run only on a particular domain. As the real domain of WSC problem is the whole web in practice, there is only a limited knowledge available [26]. The best approach to provide scalability and domain independency at the same time is to use a domain-independent planner together with a pre-filtering approach that filters out the irrelevant services according to the user request.

2.1.5 In-memory Databases (IMDBs) and Oracle TimesTen IMDB

An in-memory database (IMDB) is a database management system that primarily relies on the main memory for data storage [27]. Whereas the conventional disk-optimized database systems (DRDBs) are optimized for disk storage mechanisms, IMDBs use different optimizations to structure and organize data in the physical main memory. As the data reside in memory IMDBs provide much better response times and transaction throughputs when compared to DRDBs. This is important

especially for real-time applications where transactions have some deadlines to complete [28].

Most modern DRDBs make use of in-memory caches to allow rapid access and update of the data that is frequently used. [28] describes the difference between DRDBs with very large caches and IMDBs as follows: As the index structures of DRDBs are designed for disk access (e.g. B-trees), DRDBs cannot take the full advantage of memory, even if the data reside in memory. Furthermore, applications using DRDBs may have to access data through a buffer manager which computes the disk address of the data and then checks if the data is in memory. On the other hand, IMDBs always use memory addresses to refer to data.

As the storage medium of IMDBs is volatile, the system loses all the stored information in case of power failure or reset. Because of this problem, IMDBs can be said to lack support for durability. The mechanisms used by IMDBs to provide durability is as follows [27]:

- **Snapshot files**, that records the database state at a given time. These files are normally generated on request or when IMDB does a controlled shut-down. This mechanism can only provide partial durability as it is still possible to lose data in case of a power failure or system crash.

- **Transaction logging,** which records the database changes in a journal file and facilitates automatic recovery of an in-memory database.

- **Non-volatile RAM,** which is usually an electrically erasable programmable ROM (EEPROM) or a static RAM backed up with battery power. With the help of this storage, the data store can be recovered from its last consistent state after reboot.

- **High availability implementations,** that makes use of database replication for automatic failover to an identical stand-by database in case of primary database failure. To prevent data loss in case of a complete system crash,

IMDB replication is generally used in conjunction with one or more of the above mechanisms.

Oracle TimesTen IMDB:

Oracle TimesTen In-Memory Database (IMDB) is a memory-optimized relational database that fits entirely in the physical memory. It is persistent and recoverable and the access is provided via standard SQL interfaces. TimesTen IMDB is maintained in the operating system's shared memory segments and contains all user data, indexes, system catalogs, log buffers, lock tables, and temp space. Figure 2.7 which is adapted from [30] shows the components of TimesTen IMDB.

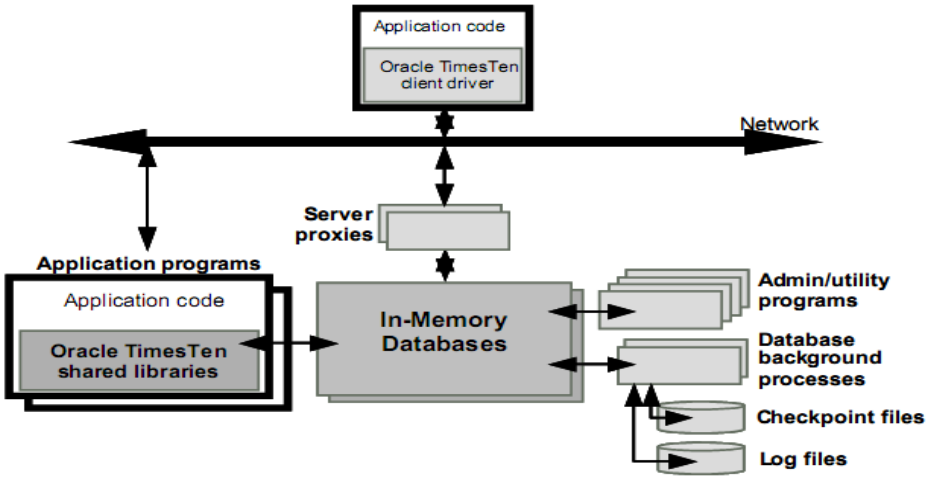

Figure 2.7: Components of TimesTen

The components are briefly described as follows [30]:

Background Processes: Provide services for startup, shutdown and application failure detection at the system level, and provide loading checkpointing and deadlock handling at the database level.

Administrative Programs: Invoked by users to perform services like backup/restore, database migration and system monitoring.

Checkpoint and log files: These files are stored on permanent disk and changes to the database and transaction logs are written to these files periodically. In case a recovery is required, TimesTen merges the database checkpoint on disk with the completed transactions that are still in the log files.

TimesTen ensures full conformance to ACID properties. The techniques used in TimesTen to provide durability are quite similar to the techniques used in conventional databases. As in all transaction-oriented systems, TimesTen durability is provided via a combination of change logging and periodic refreshes of a version of the database residing on a disk [30].

2.2 Related Work

There exist some recent approaches that deal with the scalability problems in WSC systems. Most of these approaches make use of some filtering mechanisms to eliminate the unusable services. In the following part, the most important ones approaches are briefly described and compared to this work. In addition, the WSC framework to which the scalability is added in this thesis is described with its strengths and weaknesses.

2.2.1 Semantic Web Service Composition Framework Based On Parallel Processing

In [31] the authors present a WSC system that is based on three issues: Finding all possible solutions, maximizing preprocessing and parallel processing. The system finds all possible solutions for the given composition request and selects the one that provides the best quality of service (QoS). In order to allow quick responses to user queries, the system performs a huge amount of preprocessing. Furthermore, the framework is designed to maximize the utilization of multi-processor environment.

In the preprocessing phase of the system, a relational repository is used to store some data about the available services that can help to decrease the time required for the composition. The stored data consists of service input-output information, possible chains among the services as well as the list of services whose invocation depends on the initial state provided by the user. A chain between two services means the invocation of the first service provides a data which may be consumed by the second. The services that depend on the initial state are the ones for which at least one of the inputs cannot be provided by other services. Thus, the availability of the inputs depends on the initial state provided by the user. If there are other services that depend on these services for some of their inputs, these services are also stored as depending on the initial state [31].

The composition algorithm of the system is divided into three different threads which run in parallel on the same set of services: A forward chaining thread which uses the pre-evaluated chaining information to reach the goal from the given initial state, an unusable action finder thread which eliminates the initial state depending services that cannot be invoked with the given initial state, and a backward chaining thread which makes a similar execution to the forward chaining thread in the reverse direction to find the possible compositions. The final composition is created by considering the QoS parameters of the services [31].

In this thesis, a similar preprocessing approach is used with a similar forward chaining and unusable action finder threads. This thesis implements an additional unusable action finder thread which runs in the reverse direction and eliminates the actions that cannot have contribution to the given goal. Our filtering approach considers the precondition-effect predicates as well as the input-output types while finding the chains among services. In addition, since we use an in-memory database to store the preprocessed data, we achieve faster database update and query processing. Another difference of this thesis is the method of finding the final composition. [31] runs a backward chaining algorithm and considers the QoS parameters of services; however we find the final plan via Simplanner which finds the composition with minimum number of services.

2.2.2 SEMAPLAN: Combining Planning with Semantic Matching to Achieve Web Service Composition

The approach in [32] is a hybrid approach which combines semantic matching and AI planning algorithms to enable WSC in the presence of inexact terms. The system utilizes a backward searching algorithm to filter the services that are composed via a metric planner. This backward searching collects all services whose outputs match at least one of the inputs of the services in the previous level. For fast discovery of the matching services, the algorithm uses a semantic similarity map in each filtering level. The semantic similarity map is created by a semantic matcher that uses some domain-specific and domain-independent information in a pre-processing phase to find the similarities between the services.

The main advantage of this approach is to be able to find service matches in the presence of inexact terms. The related terms in domain-independent ontologies are found via some tokenization techniques and a thesaurus. In domain-dependent ontologies, these terms are found using a semantic network-based ontology management system. Different from this approach, in the preprocessing phase of this thesis, we find the input-output, precondition-effect matches of services and use this information in the filtering phase. In addition, our system does not depend on any domain knowledge for filtering.

2.2.3 A Planning Graph Based Algorithm for Semantic Web Service Composition

In [33] a forward chaining algorithm is implemented which makes use of a leveled graph called “simplified planning graph” and treats each service as a planning action. The difference of the simplified planning graph is that actions in each action pair are independent and no mutex relations exist between actions or propositions. Each web service in the given problem is mapped to an action of a planning graph by mapping the input parameters of each service to the action’s preconditions and the output parameters to the action’s effects.

The algorithm in [33] constructs the planning graph as follows: The initial state of the problem corresponds to level 0 of propositions. The following levels consist of an action level and a proposition level. Action level 1 is the set of actions whose preconditions are satisfied in level 0 and proposition level 1 is the union of propositions in level 0 and the effects of actions in action level 1. This leveling continues until it reaches the fixed point level of the graph where no new actions can be added or it reaches a proposition level which includes all the required goal parameters. Since the simplified planning graph does not include mutexes, a solution can always be found for a solvable problem even if it may contain some redundant services. Furthermore, the simplified planning graph is of polynomial size and can be constructed in polynomial time [33]. While creating a new action level, the authors employ three different strategies to prune the redundant services. It is claimed in [33] that with these strategies, they find the composition as the graph is being built and no additional processing is required to find the composition after building the graph. The first of these three methods, which is eliminating the new actions that produce a subset of already existing propositions, is observed to be the most effective pruning strategy.

The forward chaining graph we use in this thesis is quite similar to the simplified planning graph used in [33]. On the other hand, this approach does not do any pre-processing which we use to store the chaining relations and problem dependency of the services. The most important difference of our approach is that we use the graph for filtering out the unusable services and employ Simplanner to find the final plan which includes the minimum number of services possible. However, the plan is found during the graph construction in [33]. This requires additional processing to prune the redundant services. The found composition in [33] may still contain some redundant services.

2.2.4 Redundant-Free Web Services Composition Based on a Two-Phase Algorithm

The disadvantages of forward chaining and backward chaining approaches are explained for web service composition and a two-phase algorithm that makes use of both approaches is proposed to overcome the drawbacks of these approaches and find redundant-free compositions in [34]. A pre-built data structure called Link Index is used in forward-chaining phase to find the successors of services. The Link Index is a hash table that is built according to the connectivity of services. The definition of connectivity is same as the definition of chaining in [31]. Each key in the hash table corresponds to a service available in the domain and the values of each entry are the services that have connectivity with the key service. The forward chaining graph and the algorithm are very similar to the one used in [33] and this thesis. The algorithm makes use of Link Index while finding the services in each level.

The backward chaining phase is executed after forward chaining phase to eliminate the redundant services in the candidate composition. A data structure called token manager is used in backward chaining phase to record the parameters that are uniquely covered by each service [34]. In our approach we do not employ any backward chaining as we use Simplanner to find the final composition from the services filtered by the forward chaining algorithm.

2.2.5 Dynamic Planning Approach to Automated Web Service Composition

In [1], a comprehensive WSC framework which automates the composition task in terms of time and adaptability to real world environments is described. The approach is based on AI planning and the proposed framework utilizes Simplanner [6] which is specially designed for highly dynamic and nondeterministic environments. One of the main features of the frameworks is that it interleaves planning and execution. With the help of this feature and the resiliency of Simplanner to dynamic environments, when something unexpected happens like service unavailability, the

framework is able to initiate dynamic re-planning and handle the unexpected situation. Dynamic re-planning is also initiated when the executed services have some nondeterministic effects to change the current state and prevent such effects. Furthermore, this framework makes use of WS-Business Activity Framework [7] which enables the compensation of world altering effects in case of execution failure. Figure 2.8 which is adapted from [1] shows the overall architecture of this framework.

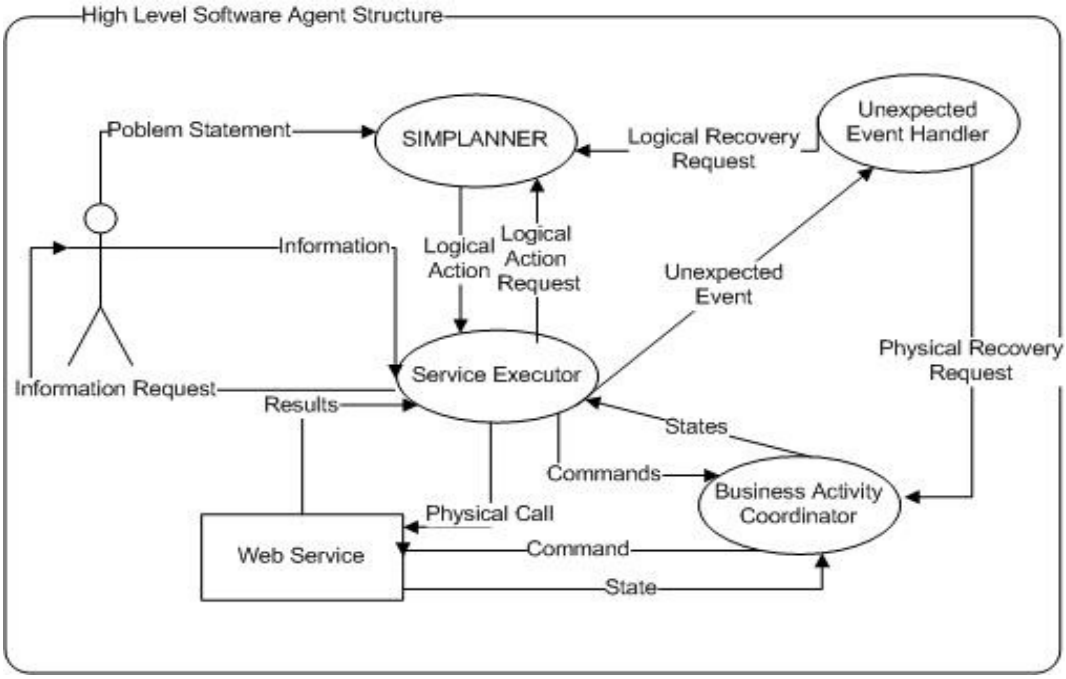

Figure 2.8: Dynamic Planning Approach Architecture

The system is composed of five phases which are briefly described as follows:

Preprocessing: In this phase the system retrieves the semantic and syntactic descriptions of services as well as the user provided domain ontology and the initial and goal states of the composition problem. OWL-S, WSDL and OWL languages are used in these descriptions. This information is used to create PDDL descriptions

which are used in the planning phase and the required service client codes which invoke the services during execution.

Planning: Almost all of the work in this phase is handled by Simplanner. Simplanner uses the PDDL descriptions created in the preprocessing phase and finds an initial plan and continues to refine the plan during the lifetime of the session. The planner sends an action to action handling/execution module one at a time to be executed. If some unexpected events occur during the execution phase, the planner is notified for re-planning.

Action Handling: This phase handles the logical action provided by the planning phase. Firstly, the real values of the logical parameters are found from the initial information provided by the user. If a real value is missing, the user is prompted and asked for it. If the user provides the asked parameter, the execution of the action takes place. Otherwise a notification is sent to the unexpected event handler which tries to resolve the problem.

Execution: The real service call is performed in this phase. If the invoked service is an information gathering service, it is done as a simple service call. On the other hand, if the service has some world altering effects, the service is called in a conformant way to WS-Business Activity and WS-Coordination specifications. The service client codes created in the preprocessing phase are used in this phase to invoke the services.

Unexpected Event Handling: This phase handles the unexpected events that may occur during service execution like network problems and missing required information. In such cases, the planner is notified in this phase for re-planning. If re-planning cannot produce a new solution the transactional operations are rolled back to prevent the side effects of the unsuccessful attempts.

The framework in [1] addresses effectively many of the open problems for web service composition mentioned in [11]. However, it lacks scalability which is a crucial property to enable the usage of the framework in real world environments

where thousands of services exist. The reason for this is that this framework utilizes a domain independent planner for the composition generation. The most suitable way to provide scalability to the systems using domain independent planners is to apply some pre-filtering to the action set before invoking the planner. In addition, this framework lacks maintaining the available services. Each framework invocation is dependent to providing the whole service set as well as the problem which decreases the practical usability of the framework significantly. The most important contributions of this thesis are to provide scalability to this framework with a filtering system and to provide service maintenance with some service storage mechanisms. These contributions are detailed in the following chapters.

CHAPTER 3

WEB SERVICE PREPROCESSING

There are two important problems in the existing WSCE framework which are its inability to scale well with the growing service domains and inability to store and maintain a large number of services. One of the main goals of this thesis is to enhance the existing WSCE framework with an effective preprocessing system that is responsible for service storage and maintenance. This preprocessing system also stores chains and dependencies of services which help to decrease the time required for the pre-filtering process that plays an important role in providing scalability to the framework. For data storage, an in-memory relational database (IMDB), namely Oracle's TimesTen IMDB and a directory structure are used. The combination of these two storage methods is called the service repository within the context of this thesis. The reason for choosing an IMDB is that IMDBs have much smaller response times to database queries when compared to disk relational databases. Database query response time has a very significant effect on the performance of the overall system because the pre-filtering process has a heavy data access and update traffic.

This chapter explains the structure of the service preprocessing system and the steps followed while adding a new service data to the service repository. In addition, the definitions of the terms *action chain* and *action dependency* are given in this chapter and it is explained how this information is stored in the service repository and how it decreases the time required for the pre-filtering process. Figure 3.1 illustrates the overall architecture of the service preprocessing system in this thesis.

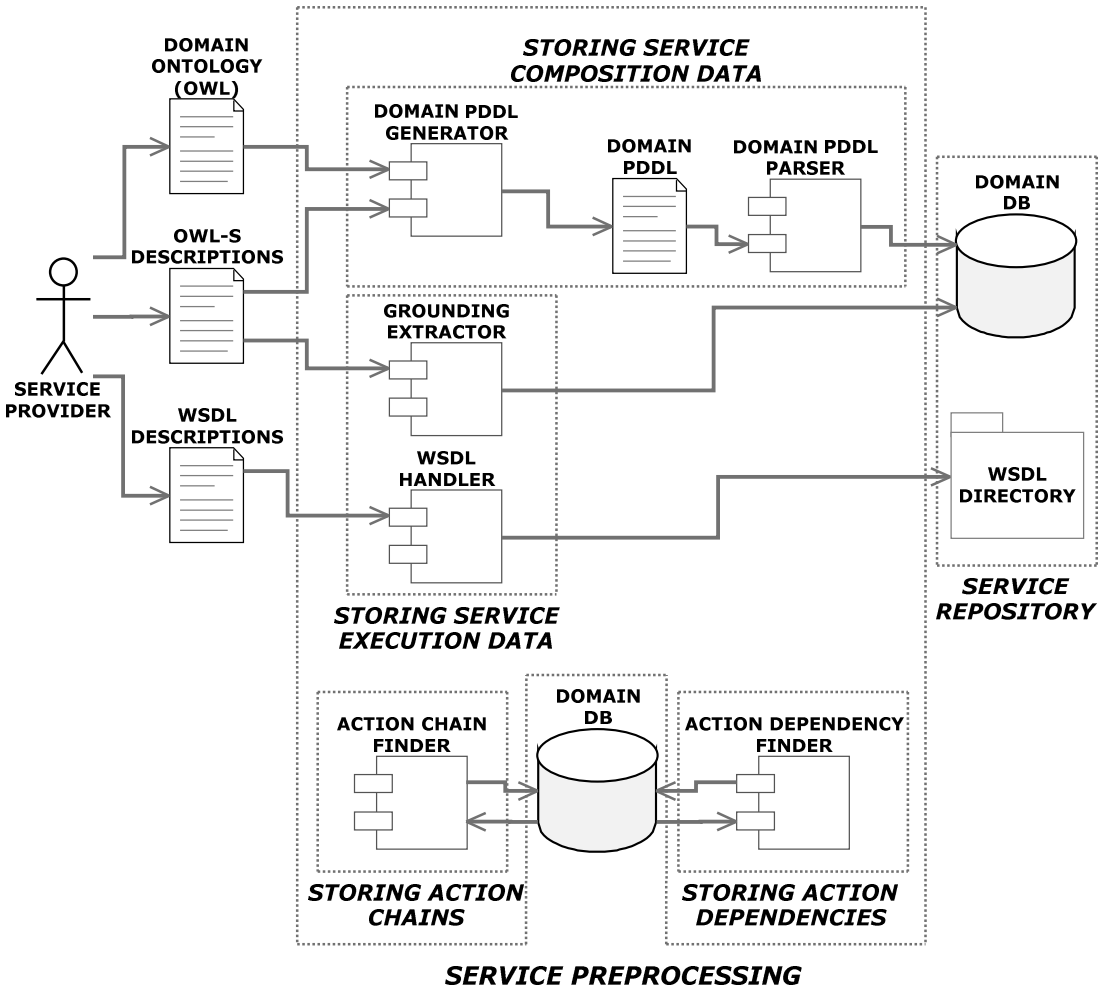

Figure 3.1: Overall architecture of Service Preprocessing system

The rest of the chapter is organized as follows: Section 1 describes the rules applied to convert OWL and OWL-S service descriptions to PDDL format. Section 2 gives details about the service information stored in the system which is used for service pre-filtering, composition and execution. Finally, Section 3 explains the details of action chains and dependencies which are used during the pre-filtering phase to select the candidate actions for composition.

3.1 OWL/OWL-S to PDDL Conversion of Services

The web service composition approaches based on AI Planning require applying an OWL/OWL-S to PDDL conversion on their input data. This is because most novel AI planners use domains and problems described in the PDDL format. On the other hand, semantic annotation of web services is mostly done with OWL and OWL-S which are widely accepted formats for the semantic domain and service annotation respectively. Since the existing WSCE framework uses an AI planner using the PDDL format, the system in thesis applies a conversion mechanism from OWL/OWL-S to PDDL.

OWL and OWL-S have many characteristics in common with PDDL which makes it straightforward to convert from one format to the other. There exist some previous works that define similar rules for this conversion [29], [39], [40]. They propose some techniques for OWL/OWL-S to PDDL conversions that are very similar to each other. The most important difference in [39] is the usage of KIF language for service precondition and effect representation which is one of the recommended languages in OWL-S 1.1 specification. In [29], a custom language namely PDDXML [29] is used for precondition and effect representation. Custom languages are also acceptable in OWL-S descriptions because the language used for preconditions and effects is not explicitly determined in the OWL-S 1.1 specification. Another difference between the approaches is the way they represent the service input-outputs with PDDL. [39] converts input and output parameters of OWL-S services directly to PDDL action parameters. [29] also makes use of the same conversion but it adds a new predicate to PDDL representation, namely agentHasKnowledgeAbout(X) which is used to show the information availability. The predicate agentHasKnowledgeAbout(X) is necessary for the WSC domain, especially for the cases where the information is partially observable.

The existing WSCE framework applies a conversion mechanism similar to the one in [29] including PDDXML language for the precondition and effect representation, with some modifications that are required to handle nondeterministic cases such as

service failures. In [1], new predicates are generated for each web service that represent their availability. This thesis implements exactly the same rules for conversion from OWL/OWL-S to PDDL with [1] because the purpose of the conversion in this thesis is to prepare PDDL descriptions of services which will be used to invoke Simplanner in [1] after the pre-filtering process described in the next chapter is completed. These conversion rules are as follows:

- OWL classes are converted to PDDL types. During this conversion the class-subclass hierarchy in OWL classes is preserved.

OWL Definition	**PDDL Definition**
<owl:Class rdf:ID="Region"/>	(:types Region – object)
<owl:Class rdf:ID="ConsumableThing"/> <owl:Class rdf:ID="PotableLiquid"> <rdfs:subClassOf rdf:resource="#ConsumableThing" /> ... </owl:Class>	(:types ConsumableThing – object PotableLiquied - ConsumableThing)

Figure 3.2: OWL class – PDDL type conversion

- OWL properties (object, datatype and functional properties) are converted to PDDL predicates.

OWL Definition	PDDL Definition
<owl:Class rdf:ID="Wine"/> <owl:Class rdf:ID="WineGrape"/> <owl:ObjectProperty rdf:ID="madeFromGrape"> <rdfs:domain rdf:resource="#Wine"/> <rdfs:range rdf:resource="#WineGrape"/> </owl:ObjectProperty>	(:types Wine – object WineGrape – object) (:predicates (madeFromGrape ?WineParameter – Wine ?WineGrapeParameter – WineGrape))

Figure 3.3: OWL property – PDDL predicate conversion

- OWL individuals, i.e. instances of OWL classes, are converted to PDDL objects.

OWL Definition	PDDL Definition
<owl:Class rdf:ID="Region"/> <Region rdf:ID="CentralCoastRegion" />	(:types Region – object) (:objects CentralCoastRegion – Region)

Figure 3.4: OWL individual – PDDL object conversion

- OWL-S service descriptions are converted to PDDL actions. The predicate valid<servicename> is added to the preconditions of each action.

OWL Definition	PDDL Definition
<service:Service rdf:ID="BookFinderService"> <service:presents rdf:resource="#BookFinderProfile"/> <service:describedBy rdf:resource="#BookFinderProcess"/> <service:supports rdf:resource="#BookFinderGrounding"/> </service:Service>	(:types Region – object) (:predicates validBookFinderService) (:action BookFinderService (:precondition (validBookFinderService)))

Figure3.5: OWL-S service – PDDL action conversion

As mentioned before, service precondition and effects are presented with the PDDXML format in OWL-S service descriptions. PDDXML [29] simply uses OWL properties and OWL-S service parameters to describe the preconditions and effects of the service.

- PDDXML service preconditions are converted to PDDL action preconditions.

PDDXML Definition	PDDL Definition
<precondition> <and> <pred name="validPersonalFlightAccount"> <param>?Person</param> <param>?AccountData</param> </pred> </and> </precondition>	(:action ServiceName :parameters (?Person - Person ?AccountData - Account) :precondition (validPersonalFlightAccount ?Person ?AccountData))

Figure 3.6: precondition – PDDL precondition conversion

- PDDXML service effects are converted to PDDL action effects.

PDDXML Definition	PDDL Definition
<effect> <and> <pred name="isBookedFor"> <param>?Flight</param> <param>?Customer</param> </pred> </and> </effect>	(:action ServiceName :parameters (?Flight - Flight ?Customer - Person) :effect (isBookedFor ?Flight ?Customer))

Figure 3.7: PDDXML effect – PDDL effect conversion

- OWL-S input and output parameters are converted to PDDL parameters. agentHasKnowledgeAbout(X) predicate is added to action preconditions for each input and to action effects for each output.

OWL-S Definition	PDDL Definition
<profile:hasInput> <process:Input rdf:ID="Flight"> <process:parameterType rdf:datatype=TravelOntology.owl#Flight </process:parameterType> </process:Input> </profile:hasInput>	(:action ServiceName :parameters (?Flight - Flight) :precondition (agentHasKnowledgeAbout ?Flight))
<profile:hasOutput> <process:Output rdf:ID="VehicleTransport"> <process:parameterType rdf:datatype=TravelOntology.owl#Transport </process:parameterType> </process:Output> </profile:hasOutput>	(:action ServiceName :parameters (?VehicleTransport - Transport) :precondition (agentHasKnowledgeAbout ?VehicleTransport - Transport))

Figure 3.8: OWL-S parameter – PDDL parameter conversion

3.2 Storing Service Information

The service repository designed for this thesis consists of a domain database which is a relational in-memory database (IMDB) and a directory structure. The stored service data is divided into two different categories: composition data and execution data. The composition data is the data required for finding the composite service and it is completely stored in the domain database. This data is created by parsing the PDDL descriptions of the services and these PDDL descriptions are obtained by applying the OWL/OWL-S-to-PDDL conversion rules described in the previous section. Execution data is the data required for executing the services involved in the composition. This data consists of the WSDL description and the logical action – physical service mapping information for each service. The WSDL descriptions are stored as text files in the directory structure (WSDL Directory in Figure 3.1) and the logical action – physical service mapping information is stored in the domain database together with the composition data of the service. The rest of this section explains the steps followed to add new service data to the repository and gives the detailed content of the composition and execution data.

3.2.1 PDDL Domain Generation

Since OWL and OWL-S are most widely used languages for the semantic annotation of web services, the preprocessing system in this thesis accepts the services annotated with these languages as input. The first step in the preprocessing phase is to convert the service information annotated with OWL/OWL-S to PDDL format. There are two reasons for this: Firstly, the existing WSCE framework uses an AI planner that requires PDDL data. Thus, PDDL action domain has to be created somewhere before invoking the planner to find the composition. Doing this conversion in the preprocessing phase prevents the time consumption during composition. In addition, since PDDL is a simpler data format when compared with OWL and OWL-S, storing PDDL information in a relational database is easier and provides space efficiency. The conversion rules given in the previous section are

applied while converting OWL/OWL-S service annotations to PDDL domain format.

3.2.2 Storing Service Composition Data

After converting the service data to PDDL domain format, this PDDL domain is parsed and stored in the tables in the domain database. For parsing PDDL domain, PDDL4J [35] is used which is an open source library implemented with Java and it is generally used in AI planners which are based on PDDL. The purpose of storing this parsed PDDL data is to use it in the pre-filtering and composition phases to find the composition with the help of Simplanner. The parsed PDDL domain and the relational model used for storage are explained in the following:

PDDL Types, Subtypes and Supertypes:

The PDDL domain format includes the types used in the domain with their supertypes. This type-supertype relation is extracted from the ontological hierarchy in OWL classes and is stored in the domain database to be used in creating the *subpredicates* and *superpredicates* of the predicates available in the PDDL domain. There are two tables in the domain database that are used for PDDL type storage: Type and HasSupertype. Figure 3.9 shows the entity-relationship (ER) diagram of these tables and Figure 3.10 shows the SQL queries used to create them.

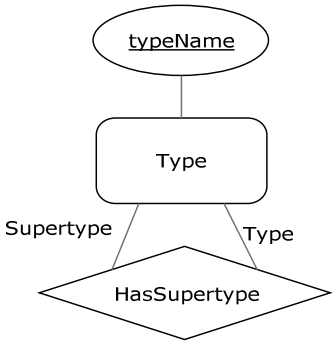

Figure 3.9: ER diagram of Type and HasSupertype

```
CREATE TABLE Type (    typeName CHAR(50),
                       PRIMARY KEY (typeName)
);
```

```
CREATE TABLE HasSupertype ( type CHAR(50),
                            supertype CHAR(50),
                            PRIMARY KEY (type, supertype),
                            FOREIGN KEY (type) REFERENCES Type
                                    (typeName) ON DELETE CASCADE,
                            FOREIGN KEY (supertype) REFERENCES
                                    Type (typeName) ON DELETE CASCADE
);
```

Figure 3.10: SQL queries for creating Type and HasSupertype tables

A part of the type definitions in a PDDL domain is shown in Figure 3.11. An example can be given as follows to illustrate how type information is stored in the database. In Figure 3.11, a supertype of type “DepartureAirport” is “Airport”. Since there is transitivity among types and supertypes, “Location” and “object” types are also supertypes of “DepartureAirport”. Similarly, “DepartureAirport” type is a subtype of each one of these supertypes. In addition, each type is a supertype and a subtype of itself. Figure 3.12 shows how type and supertype information are stored for “DepartureAirport” in the two tables.

```
…
(:types Transport - object
Location - object
Hospital - Location
Airport - Location
ArrivalAirport - Airport
DepartureAirport - Airport
Address – object
Flight - Transport
ProvidedFlight - Flight
…
)
…
```

Figure 3.11: Sample PDDL domain type definitions

Type
object
Location
Airport
DepartureAirport

Type	Supertype
DepartureAirport	object
DepartureAirport	Location
DepartureAirport	Airport
DepartureAirport	DepartureAirport

Figure 3.12: Type and HasSupertype contents for “DepartureAirport”

PDDL Predicates, Subpredicates and Superpredicates:

PDDL domains include the set of predicates used by the actions in the domain. The predicates are extracted from OWL class properties and are stored in the domain database. In addition to predicates, their subpredicates and superpredicates are also generated and stored. The following example explains the definition of the terms *superpredicate* and *subpredicate*: Assume the domain database contains two types as typeA and typeB and their subtypes and supertypes as subtypeA, subtypeB, supertypeA and supertypeB. Furthermore, assume it contains a predicate pred(typeA, typeB). In this case, pred has four subpredicates including itself (i.e. pred(typeA, typeB)) and pred(typeA, subtypeB), pred(subtypeA, typeB), pred (subtypeA, subtypeB). In addition, it has four super-predicates as pred(typeA, supertypeB), pred(supertypeA, typeB), pred(supertypeA, supertypeB) and itself. As the example implies, each predicate is both a subpredicate and superpredicate of itself. The stored subpredicates and superpredicates are used in the pre-filtering phase which considers the predicate hierarchies during action selection.

There are two tables in the database that are used for the predicate information: Preds and HasSuperpreds. Figure 3.13 shows the ER diagram of these tables and Figure 3.14 shows the SQL queries used to create the tables.

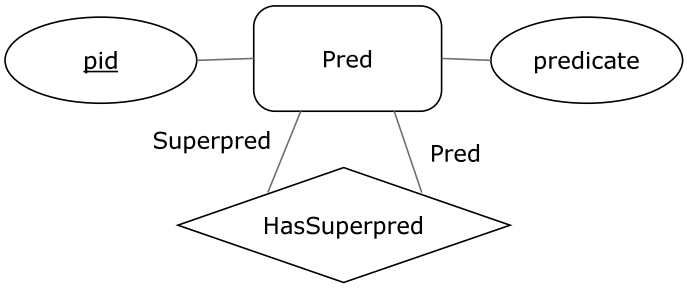

Figure 3.13: ER diagram of Pred and HasSuperpred

```
CREATE TABLE Pred (     pid int NOT NULL,
                        predicate CHAR(150) UNIQUE NOT NULL,
                    PRIMARY KEY (pid)
);
```

```
CREATE TABLE HasSuperpred ( pid int,
                            superpid int,
                            PRIMARY KEY (pid, superpid),
                            FOREIGN KEY (pid) REFERENCES Pred
                                (pid) ON DELETE CASCADE,
                            FOREIGN KEY (superpid) REFERENCES Pred
                                (pid) ON DELETE CASCADE
);
```

Figure 3.14: SQL queries for creating Pred, HasSuperpred tables

Figure 3.15 shows a part of the predicate definitions of a sample PDDL domain. An example can be given as follows to illustrate how the predicate information is stored in the domain database: In Figure 3.15, the predicate hasNearestAirport() has two parameters of types Address and Airport. As can be seen from Figure 3.11, Address has one supertype, object and it has no subtypes. Airport has two supertypes, Location and object and two subtypes, DepartureAirport and ArrivalAirport. Therefore, hasNearestAirport(Address, Airport) has six superpredicates as hasNearestAirport (object, Airport), hasNearestAirport(object, Location), hasNearestAirport(object, object), hasNearestAirport(Address, Location), hasNearestAirport(Address, object) and hasNearestAirport(Address, Airport). In

addition, it has three subpredicates as hasNearestAirport(Address, DepartureAirport), hasNearestAirport(Address, ArrivalAirport) and hasNearestAirport(Address, Airport). Figure 3.16 shows how the predicate and superpredicate information is stored for hasNearestAirport() for the two tables mentioned above.

```
…
(:predicates
(personalProvidedTransport ?VehicleTransport_range_parameter - VehicleTransport )
(validPersonalTransportAccount ?Account_range_parameter - Account )
(validMedicalTransportAccount ?Account_range_parameter - Account )
(hasNearestAirport ?Address_domain_parameter - Address ?Airport_range_parameter -
Airport )
…
)
…
```

Figure 3.15: Sample PDDL domain predicate definitions

pid	predicate
1	hasNearestAirport (Address, Airport)
2	hasNearestAirport (object,Airport)
3	hasNearestAirport (object, Location)
4	hasNearestAirport (object, object)
5	hasNearestAirport (Address,Location)
6	hasNearestAirport (Address, object)
7	hasNearestAirport (Address, DepartureAirport)
8	hasNearestAirport (Address, ArrivalAirport)

pid	superpid
1	1
1	2
1	3
1	4
1	5
1	6
7	1
8	1

Figure 3.16: Pred and HasSuperpred contents for hasNearestAirport(Address, Airport)

PDDL Actions, Effects and Preconditions:

The most important part of the PDDL domain description is the action definitions. The action definitions are created from the OWL-S service descriptions and each action definition contains all semantic information available in the corresponding service. The preconditions of actions are extracted from the services' inputs and preconditions, and the effects of actions are extracted from the services' outputs and effects. The action creation is done according to the rules described in Section 3.1. The stored actions are used in both the pre-filtering process and the planning phase to find the composite service which is a plan of actions basically.

There are three tables in the database that are used to store te action information: Action, HasPrec and HasEffect. The Action table stores the name and physical mappings of the actions and HasPrec and HasEffect store the predicates that correspond to the preconditions and effects of the actions, respectively. Figure 3.17 shows the ER diagram of these tables and Figure 3.18 shows the SQL queries used to create them. The isInvokable attribute of the Action table is accessed and updated during the pre-filtering process and its use will be explained in the next chapter. It can normally take only two values, 0 and 1. When a new action is added to the domain database, this attribute is set to 1 as default for the newly added action. The mappingXml attribute of the Action table is set during the phase of storing the execution data and its details are explained in Section 3.2.3.

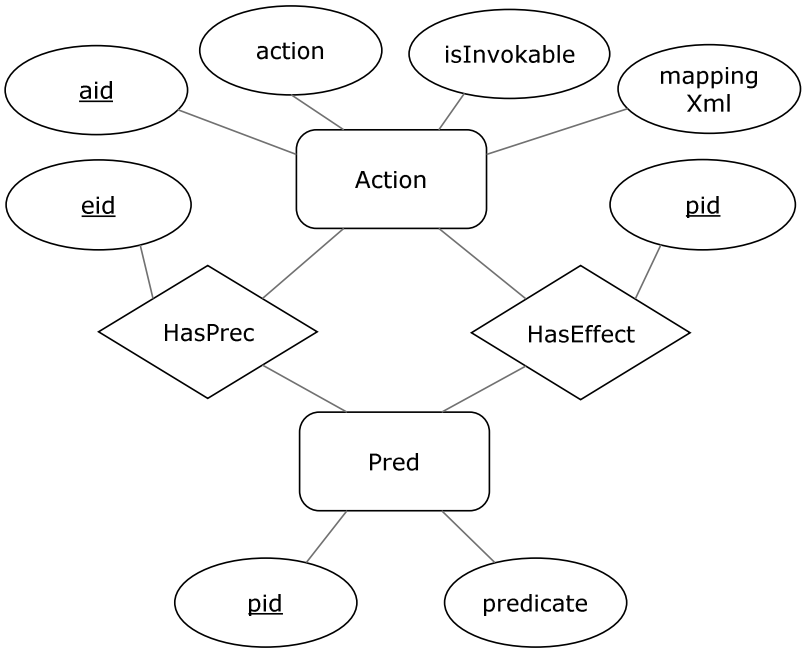

Figure 3.17: ER diagram of Action, HasPrec and HasEffect

```
CREATE TABLE Action (  aid int,
                       action CHAR(50) UNIQUE NOT NULL,
                       isinvokable int NOT NULL,
                       mappingXml CHAR(200) NOT NULL,
                       PRIMARY KEY (aid)
);
```

```
CREATE TABLE HasPrec (      pid int NOT NULL,
                       aid int NOT NULL,
                       predid int NOT NULL,
                       PRIMARY KEY (pid),
                       FOREIGN KEY (aid) REFERENCES
                            Action(aid) ON DELETE CASCADE,
                       FOREIGN KEY (predid) REFERENCES
                            Pred(pid) ON DELETE CASCADE
);
```

```
CREATE TABLE HasEffect (    eid int NOT NULL,
                            aid int NOT NULL,
                            predid int NOT NULL,
                            PRIMARY KEY (eid),
                            FOREIGN KEY (aid) REFERENCES
                            Action(aid) ON DELETE CASCADE,
                       FOREIGN KEY (predid) REFERENCES
                       Pred(pid) ON DELETE CASCADE
);
```

Figure 3.18: SQL queries for creating Actions, HasPrec, HasEffect tables

Figure 3.19 shows the definition of an action from a sample PDDL domain. This action is stored in the domain database as follows: Assume the three preconditions of this action, which are agentHasKnowledgeAbout(Account), agentHasKnowledgeAbout (Creditcard) and agentHasKnowledgeAbout(Person) are stored in the Pred table with pids 10, 11 and 12 respectively. In addition, assume the effect of the action, validPersonalTransportAccount(Account) is stored in the Pred table with pid 13. With these assumptions, the contents of the related tables are illustrated in Figure 3.20 when this action is stored in the domain database.

```
…
(:action CreateVehicleTransportAccountAtomicProcess

:parameters ( ?DesiredAccountData - Account ?CreditcardInformation - Creditcard
?UserData - Person)

:precondition (and (agentHasKnowledgeAbout ?DesiredAccountData)
(agentHasKnowledgeAbout ?CreditcardInformation)
(agentHasKnowledgeAbout ?UserData)
)

:effect (and (validPersonalTransportAccount ?DesiredAccountData)
))
…
```

Figure 3.19: Sample PDDL domain action definition

aid	action	IsInvokable	mapping Xml
1	CreateVehicle TransportAccount AtomicProcess	1	<action> … </action>

pid	aid	predid
1	1	10
2	1	11
3	1	12

eid	aid	predid
1	1	13

Figure 3.20: Action, HasPrec and HasEffect contents for CreateVehicleTransportAccountAtomicProcess

3.2.3 Storing the Service Execution Data

The existing WSCE framework enables automatic service execution as well as service composition. In order to create the service execution environment, the existing WSCE framework requires the syntactic descriptions of the services such as syntactic types of operation arguments, service end point, used communication protocol in addition to the semantic annotations. By using these syntactic descriptions, [1] creates client stubs and logical action - physical service mappings to execute the physical counterparts of the logical actions found by Simplanner during composition generation. The client stubs are generated by the client stub generator component [1] which uses the WSDL descriptions of the available

services and WSDL2JAVA tool of Apache Axis [41]. The logical action – physical service mappings are generated by the grounding extractor component [1] which uses the grounding sections of OWL-S descriptions of the available services. Since this thesis includes the OWL-S description processing as a part of the enhancements to the existing WSCE framework, the grounding extractor component is also included in the scope of this thesis. With this change, the existing WSCE framework is directly provided with the logical action – physical service mappings instead of semantic service descriptions.

In this thesis, two types of service execution data are stored: The first one is the set of WSDL descriptions of the services which are required by the existing WSCE framework to create the client stubs. WSDL descriptions are stored in a directory structure as separate WSDL files. The other type of service execution data is logical action – physical service mappings mentioned above. The information included in these mappings is completely extracted from the grounding sections of OWL-S descriptions and the purpose of constructing these mappings is to make it easier for the existing WSCE framework to process this information. The grounding extractor component uses the following parts of OWL-S grounding section to create the mappings:

- **wsdlDocument:** Represents the URI of service WSDL.
- **wsdlOperation:** Represents the URI of WSDL operation.
- **wsdlInputMessage:** Represents the URI of WDSL message definition that carries the inputs of the process.
- **wsdlInput:** Represents the mapping between OWL-S input parameters and WSDL counterparts.
- **wsdlOutputMessage:** Represents the URI of WSDL message definition that carries the outputs of the process.

- **wsdlOutput:** Represents the mapping between OWL-S output parameters and their WSDL counterparts.

```
<action name= “LogicalOperation” wsdlOperation=”PhysicalOperation” />
        <inputs>
                <input wsdlName = “PhysicalInput” owlsName = “LogicalInput” />
                 .....
        </inputs>
        <outputs>
             <output wsdlName = “PhysicalOutput” owlsName = “LogicalOutput” />
                        …….
        </outputs>
</action>
```

Figure 3.21: Logical action – physical service mapping structure

The format of the mappings created by the grounding extractor for each service is shown in Figure 3.21. The xml structure illustrated in the figure is created for each available service in the service repository and stored in the mappingXml attribute of the Action table (Figures 3.18 and 3.19). When the pre-filtering phase finds the candidate services to be included in the composition, WSDL descriptions of the candidate services are selected from the WSDL directory and provided to the existing WSCE framework. The client stub generator component [1] of the framework uses these WSDL descriptions to generate the client codes for the automatic service execution. In addition, an xml file named OWLS-WSDLMapping.xml is created which includes the mapping strings of these candidate services. This xml file is provided to the existing WSCE framework to be used by the OWL-S action dynamic code mapper component [1] together with the client codes generated by the client stub generator.

3.3 Storing Action Chains and Dependencies

In order to decrease the time required for the pre-filtering phase, it is required to preprocess the service domain and find and store the problem-independent data that will speed up the pre-filtering phase. This approach has a drawback that it uses additional space to store the data which decreases the pre-filtering time. However, since it has a significant contribution to the timely response of the overall system, this drawback can be ignored. In this thesis, two types of problem-independent information are extracted from the service domain, namely, action chains and action dependencies. The idea of storing this chain and dependency information is adapted from [31]. The main difference is that, in [31], this information is extracted from the input-output parameters of the services. However, in this thesis, the preconditions and effects of the PDDL actions are used. This change helps to cover the services that have defined preconditions and effects as well as the services that have only input and output descriptions. In addition, as well as storing the action dependencies to the problem initial state as in [31], in this thesis, the dependencies of actions to the problem goal state are also extracted from the PDDL domain and stored. The details of the information stored are explained as follows:

3.3.1 Storing Action Chains

The available actions in the domain database are traversed to find the possible chains between them. An effect predicate eA of an action ActionA and a precondition predicate pB of an action ActionB are said to be chained if eA is a subpredicate of pB, so ActionA's eA effect can be used to satisfy ActionB's pB precondition. In this case, ActionA and ActionB are defined as chained actions. For example, considering the type definitions in Figure 3.11, the two actions in Figure 3.22, ProposeFlightAtomicProcess and BookFlightAtomicProcess are chained because personalProvidedTransport(Flight) effect of ProposeFlightAtomicProcess is a subpredicate of personalProvidedTransport (Transport) precondition of BookFlightAtomicProcess.

(:**action** ProposeFlightAtomicProcess

:**parameters** (?DepartureAirport - DepartureAirport ?ArrivalAirport – ArrivalAirport
?RequestParameters - FlightParameters ?ProposedFlight - Flight)

:**precondition** (and (agentHasKnowledgeAbout ?DepartureAirport)
(agentHasKnowledgeAbout ?ArrivalAirport)
(agentHasKnowledgeAbout ?RequestParameters)
)

:**effect** (and (agentHasKnowledgeAbout ?ProposedFlight)
(hasDepartureLocation ?ProposedFlight ?DepartureAirport)
(hasDestinationLocation ?ProposedFlight ?ArrivalAirport)
(hasParameters ?ProposedFlight ?RequestParameters)
(personalProvidedTransport ?ProposedFlight)
))

(:**action** BookFlightAtomicProcess

:**parameters** (?Customer - Person ?AccountData - Account ?Transport - Transport)

:**precondition** (and (validBookFlightAtomicProcess)
(agentHasKnowledgeAbout ?Customer)
(agentHasKnowledgeAbout ?AccountData)
(agentHasKnowledgeAbout ?Flight)
(validPersonalFlightAccount ?AccountData)
(personalProvidedTransport ?Transport)
)

:**effect** (and (isBookedFor ?Flight ?Customer)
(medicalProvidedFlight ?Flight)
))

Figure 3.22: Example of chained actions

There is a table in the domain database that stores the chaining information between services: HasChain. This table stores the precondition and effect ids that constitute a chain for the corresponding actions. Figure 3.23 shows the ER diagram for HasChain relation and Figure 3.24 shows the SQL query used to create HasChain table in the domain database.

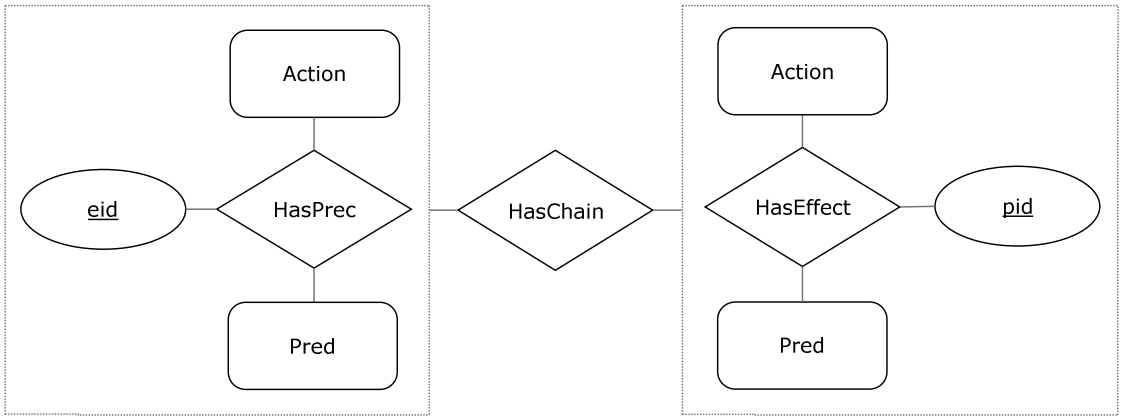

Figure 3.23: ER diagram of HasChain relation

```
CREATE TABLE HasChain (       eid int,
                              pid int,
                                    PRIMARY KEY (eid, pid),
                              FOREIGN KEY (eid) REFERENCES
                                    HasEffect(eid) ON DELETE CASCADE,
                              FOREIGN KEY (pid) REFERENCES
                                    HasPrec(pid) ON DELETE CASCADE
);
```

Figure 3.24: SQL query to create HasChain table

When the PDDL action description of a new service is added to the domain database, its preconditions and effects are added to Preds, HasPrec and HasEffect tables. Then the preconditions of the new action are compared with the effects of other actions in the HasEffect table. If one of the preconditions of the new action is a superpredicate of an effect in HasEffect table, a new record is added to HasChain table with the ids of the new action precondition and the chained effect. Similarly, the effects of the new action are compared with the preconditions of other actions in the HasPrec table. If one of the effects of the new action is a subpredicate of a precondition in HasPrec table, a new record is added to the HasChain table with the ids of the new action effect and the chained precondition.

The chaining information is quite important for decreasing the time required for the pre-filtering phase. In the pre-filtering phase, HasChain table is frequently queried to find the actions that can be chained with the set of available actions. If the domain database was not preprocessed and chains were not stored in a table, all actions in the domain database would have to be processed during the pre-filtering phase to find a match for each effect of each action in the set of available actions. With the help of HasChain table, the actions whose preconditions match to the effects of actions in the set of available actions can be gathered with a single SQL query by querying a single table quickly. The details of the pre-filtering phase are explained in the next chapter.

3.3.2 Storing Action Dependencies

Another type of problem-independent information that contributes to decreasing the time required for the pre-filtering process is action dependencies. In this thesis, the actions stored in the domain database are classified into 5 different types. This classification is done based on the chain availability for the actions' preconditions and effects. The action types are as follows:

- **Initial state dependent actions (ISDAs):** These actions are defined by adapting the term "user-data dependent service" in [31] to the PDDL terminology as follows: An action with a precondition predicate *p*, is called an *initial state dependent action* if there is no action in the domain database one of whose effects can be chained with *p*. In such a case this action can be invoked if and only if the initial state of the problem provided by the user satisfies the precondition *p*. As a result, if the initial state of the composition problem does not include a match for *p* or one of its subpredicates, this action cannot be selected as a candidate action by the pre-filtering process for the composition.

- **ISDA dependent actions:** An action is an *ISDA dependent action* if an ISDA is the only action that satisfies one of the preconditions of this action.

Furthermore, if some other action has the same dependency on an ISDA dependent action, this action is also defined as ISDA dependent action and this definition continues in a recursive manner. For example, assume three of the actions available in the domain database are ActionA, ActionB, ActionC. Assume that ActionA is an ISDA. In addition, a precondition pB of ActionB can be chained with an effect of ActionA and there is no other action in the domain database one of whose effects can be chained with pB. In this case ActionB is defined as an ISDA dependent action depending on ActionA. Furthermore, assume ActionC has a precondition pC that can be chained with an effect of ActionB and there is no other action in the domain database one of whose effects can be chained with pC. With this assumption, ActionC is also defined as an ISDA dependent action depending on ActionA, because its precedent action, ActionB, depends on ActionA. In this example, for a given problem, the invocation of ActionC and ActionB is possible only if the invocation of ActionA is possible and the invocation of ActionA is possible only if the initial state of the composition problem satisfies all of its preconditions.

- **Goal state dependent actions (GSDAs):** In this thesis, the definition of the term "user-data dependent service" in [31] is extended to cover the dependencies of actions to the composition problem goal state. An action in the domain database is a *goal state dependent action* if none of its effects predicates or their can be chained with any preconditions of the other actions in domain database. In other words, an action is a goal state dependent action if none of its effect predicates or their superpredicates is a precondition of another action in domain database.

- **GSDA dependent actions:** The definition of ISDA dependent action is adapted for the problem goal state as follows: An action is a *GSDA dependent action* if all the actions whose preconditions can be chained with the effects of this action are GSDAs or *GSDA dependent actions*. In addition,

if another action has the same dependency for a GSDA dependent action, this action is also defined as GSDA dependent action and this definition continues in a recursive manner. For example, assume three of the actions existing in the domain database are ActionA, ActionB, ActionC; and ActionC is a GSDA. If all of the effects of ActionB can be chained with the preconditions of ActionC and there is no other action in the domain database whose any precondition can be chained with any effect of ActionB, ActionB is defined as a GSDA dependent action depending on ActionC. Furthermore, assume all of the effects of ActionA can be chained with the preconditions of ActionB and there is no other action in the domain database whose some precondition can be chained with some effect of ActionA. With this assumption, ActionA is also defined as a GSDA dependent action depending on ActionC since its successor action, ActionB depends on ActionC. In this example, for a given composition problem, the invocation of ActionA and ActionB is possible only if the invocation of ActionC is possible and the invocation of ActionC is possible only if the composition problem goal state includes at least one of the effects of ActionC or their superpredicates.

- **Problem state independent Actions (PSIA):** The actions that cannot be classified with the four types described above are defined as the *problem state independent actions*. All of the preconditions of a problem state independent action can be chained with at least one of the effects of other actions in the domain database. Moreover, at least one of the effects of a problem state independent action can be chained with at least one of the other actions' preconditions.

There are four tables in the domain database that are used to store the action dependencies: HasInitDependency, HasActionDependencyInit, HasGoalDependency, HasActionDependencyGoal. HasInitDependency table stores information of ISDAs. The action id of an ISDA is stored in this table together with the predicate id of its precondition that causes the dependency to the problem initial state. HasActionDependencyInit stores the action ids of ISDA dependent actions

together with the action ids of ISDAs they are depended on. HasGoalDependency table stores the action ids of GSDAs and HasActionDependencyGoal table stores action ids of GSDA dependent actions together with the action ids of GSDAs they depend on. Figure 3.25 shows the ER diagram of the tables and Figures 3.26 and 3.27 show the SQL queries used to create these tables.

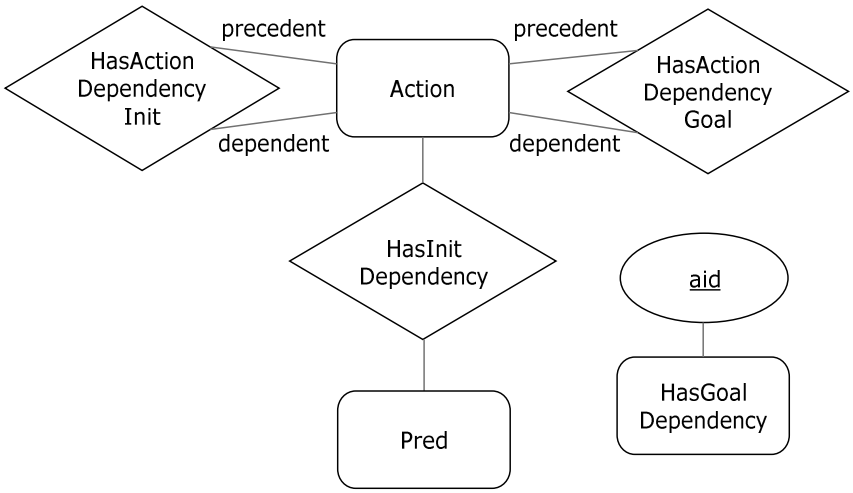

Figure 3.25: ER diagram of HasInitDependency, HasActionDependencyInit, HasGoalDependency and HasActionDependencyGoal

```
CREATE TABLE HasInitDependency (
          aid int,
        pid int,
        PRIMARY KEY (aid, pid),
        FOREIGN KEY (aid) REFERENCES
        Action(aid) ON DELETE CASCADE,
        FOREIGN KEY (pid) REFERENCES
        HasPrec(pid) ON DELETE CASCADE
);

CREATE TABLE HasActionDependencyInit (
          aid int,
          daid int,
        PRIMARY KEY (aid, daid),
        FOREIGN KEY (aid) REFERENCES
          Action(aid) ON DELETE CASCADE,
        FOREIGN KEY (daid) REFERENCES
          Action(aid) ON DELETE CASCADE
);
```

Figure 3.26: SQL queries to create HasInitDependency and HasActionDependencyInit

```
CREATE TABLE HasGoalDependency (
          aid int,
        PRIMARY KEY (aid),
        FOREIGN KEY (aid) REFERENCES
        Action(aid) ON DELETE CASCADE
);

CREATE TABLE HasActionDependencyGoal (
          aid int,
        daid int,
        PRIMARY KEY (aid, daid),
        FOREIGN KEY (aid) REFERENCES
        Action(aid) ON DELETE CASCADE,
        FOREIGN KEY (daid) REFERENCES
        Action(aid) ON DELETE CASCADE
);
```

Figure 3.27: SQL queries to create HasGoalDependency, HasActionDependencyGoal

After the PDDL description of a new action is added to the domain database and the chains of this new action are stored, the preconditions of the action are checked to find out whether all of them are chained with the effects of some other actions. If at least one of the predicates remains unchained, this action becomes an ISDA and the id of this action is stored in the HasInitDependency table together with the predicate id of the unchained precondition. Otherwise, HasInitDependency and HasActionDependencyInit tables are checked to find out whether at least one of the actions whose effects are chained with a precondition of this new action is an ISDA or an ISDA dependent action and there is no other action whose effect can be chained with this precondition. In such a case, the id of this new action is also stored in HasActionDependencyInit table since it becomes an ISDA dependent action.

After the initial state dependency checking, the new action is examined for the goal state dependencies. The effects of the new action are checked to find out whether they are chained with the preconditions of some other actions. If it is found that none of the effects are chained, this action becomes a GSDA and the id of this action is stored in the HasGoalDependency table. Otherwise, HasGoalDependency and HasActionDependencyGoal tables are checked to find out whether the actions whose preconditions are chained with the effects of this new action are GSDA or GSDA dependent action. If all of these actions are found to be GSDA or GSDA dependent action, the new action is stored in the HasActionDependencyInit table because it becomes a GSDA dependent action.

CHAPTER 4

CREATING THE WEB SERVICE COMPOSITION AND EXECUTION DATA

When the user provides the composition problem to the existing WSCE framework, the framework gives all available actions in the domain to a domain independent AI planner together with the problem description. This approach fails to be applicable when the number of actions in the domain is large because the domain independent AI planners do not scale well with the increasing domain size. This thesis provides a solution to this scalability problem by applying a pre-filtering mechanism and selecting the candidate actions that can be used to find a composite service for the provided problem. In addition to finding the candidate actions, the data required by the existing WSCE framework for finding the composite service and executing it, is prepared and provided to this framework.

This chapter explains the structure of the system that generates the service composition and execution data and the steps followed to generate this data. In addition, the rules to convert OWL composition problem to PDDL planning problem are also defined in this chapter. Figure 4.1 shows the overall architecture of the composition and execution data creation system. This system retrieves the problem description from the user in OWL format and returns the PDDL problem description, the domain description including only the filtered candidate actions, and the service execution data of the candidate actions. The rest of this chapter is organized as follows: Section 4.1 defines the rules applied for converting the OWL problem description to PDDL problem format and Section 4.2 explains the steps followed to select the candidate services and create the service composition and execution data.

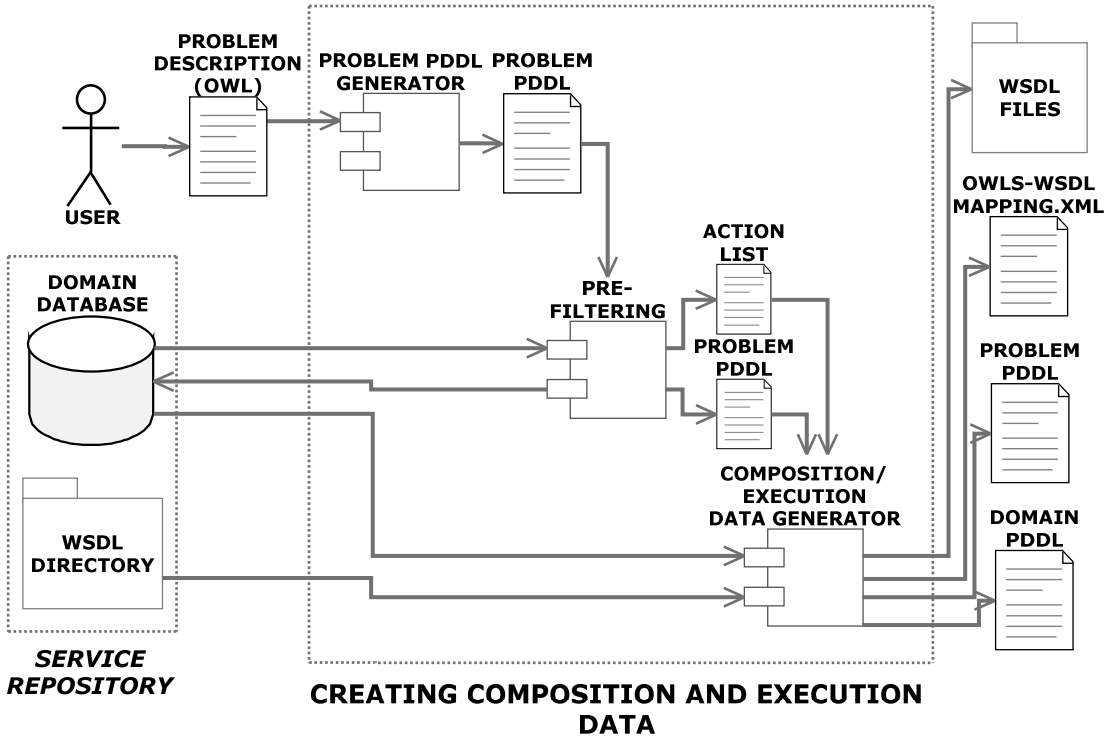

Figure 4.1: Overall architecture of composition and execution data creation system

4.1 OWL to PDDL Conversion of Composition Problem

In the existing WSCE framework the user provides the composition problem (i.e. initial and goal states) in OWL format. Since the planner in [1] uses the PDDL format for domain and problem description, the OWL problem should be converted to the PDDL format. In [1], this conversion is done during the preprocessing phase and the created problem PDDL description is provided to Simplanner together with the domain PDDL description. Since a pre-filtering process takes place before the invocation of the planner in this thesis, OWL-to-PDDL problem conversion is done before this pre-filtering process. The conversion is adapted from [1] as follows.

When the user provides the problem, it is assumed that all the required information can be provided by the user except for the information asked in the problem goal that requires information gathering. In such a case, the user explicitly states that the information asked in the problem goal is unknown and intends to learn that information. For all the defined object instances excluding the ones for which the user explicitly states unavailability, "agentHasKnowledgeAbout(obj)" predicate is added to the PDDL problem initial state. In addition, it is assumed that all services are able to operate initially and for all services, "valid<ServiceName>" predicates are added to the problem initial state. Figure 4.2 shows an example OWL problem definition and its PDDL counterpart.

OWL Definition	**PDDL Definition**
Initial State: <VehicleTransport rdf:ID = "TransportToHospital"/> <Patient rdf:ID = "Patient_0"/> **Goal State:** <VehicleTransport rdf:ID = "TransportToHospital"/> <Patient rdf:ID = "Patient_0"/> <VehicleTransport rdf:resource="#TransportToHospital"> <isBookedFor rdf:resource="#Patient_0"/> </VehicleTransport>	(:types Region – object) (:objects TransportToHospital – VehicleTransport Patient_0 – Patient) (:init (validService1) (validService2) …… (agentHasKnowledgeAbout (TransportToHospital)) (agentHasKnowledgeAbout(Patient_0)) ……) (:goal (and (isBookedFor TransportToHospital (Patient_0)))))

Figure 4.2: OWL and PDDL problem definitions

In this example, the user represents the problem in OWL by defining some logical objects and the desired state about the logical objects. If the user requests information but not a state change, a logical object "obj" is defined as in the example above and "agentHasKnowledgeAbout(obj)" is added to the definition of goal state.

In this case “agentHasKnowledgeAbout(obj)” is removed from the initial state definition.

4.2 Selecting and Generating the Composition and Execution Data

This section explains the three components illustrated in Figure 4.1 and the data flow between them.

4.2.1 PDDL Problem Generation

In [1], OWL-to-PDDL problem conversion is done during the preprocessing phase together with the conversion of OWL-S service descriptions. The rules described in Section 4.1 are used for this conversion. In this thesis, the same rules are used for the problem conversion. However, the problem conversion is now independent from the domain conversion and the domain conversion is done in the service pre-processing phase described in the previous chapter. The problem PDDL Generator component is shown in Figure 4.1 which takes OWL problem description as input and produces the PDDL counterpart of the same problem by applying the rules described in Section 4.1.

4.2.2 Action Pre-filtering

After the problem PDDL description is created, this description is used for selecting the PDDL actions from the domain database that can be used to find a composition for the PDDL problem. The action selection process is called *pre-filtering* and it is done by considering the problem initial and goal states. The actions selected from the domain database are called candidate actions and this subset of domain actions constitutes the new domain of the planning problem. Figure 4.1 shows the interaction of the Pre-filtering component with the other components and the service repository.

The algorithm used in this component consists of three different processes that run concurrently and independent of each other. These processes are designed to get the

maximum utilization from hardware platforms involving multiple processors. This is achieved by creating a multi-threaded system which creates and runs different Java threads for these three processes and their subtasks. Since all of the recent Java Virtual Machine (JVM) versions efficiently handle the work of assigning different threads to different available processors, the created threads run on different processors on a multi-processor environment. Figure 4.3 shows the three processes running concurrently in the Pre-filtering component.

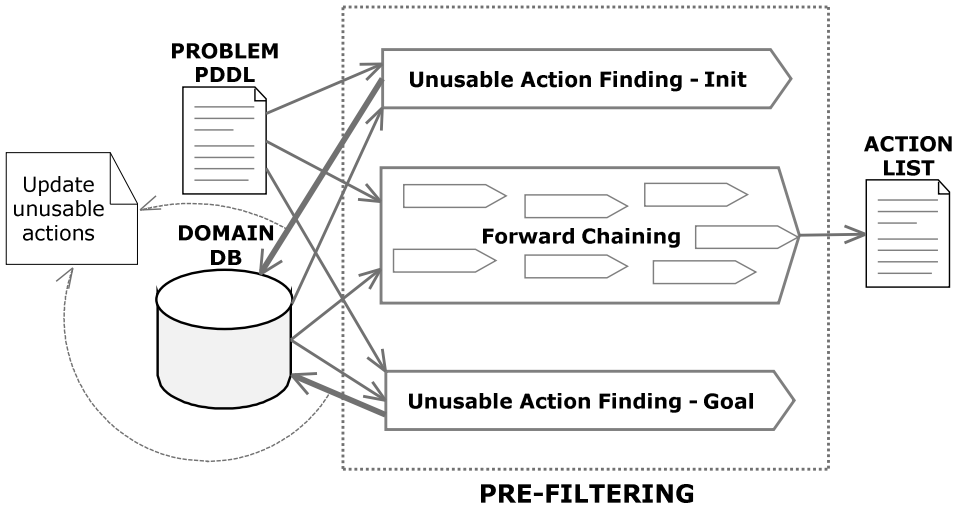

Figure 4.3: Pre-filtering component

Unusable Action Finding – Init:

This process retrieves the initial state from the problem PDDL description and checks the initial state dependent actions (ISDAs) and ISDA dependent actions. This process works in a similar way to the unusable action finder thread explained in [31]. When the user submits the problem to the system, all ISDAs and ISDA dependent actions are assumed to be usable for the submitted problem and their isInvokable attributes in the Action table are set to 1 by default in the pre-processing phase which shows that the actions can be used for a given problem and they do not

have any dependency to problem initial state or their dependencies to the initial state are satisfied for this problem. This process firstly traverses the ISDAs stored in HasInitDependency table in the domain database to check if their preconditions that cause the dependency to the initial state are satisfied. This check is simply done by comparing these preconditions with the predicates in the initial state of the problem. During this comparison super/sub-predicate relations are also considered. If it is found that the problem initial state does not include a subpredicate of an action's predicate that cause the action's dependency, isInvokable attribute of the action in Action table is set to 0 which shows that the action cannot be used to solve the given problem. This updated isInvokable attribute is checked by the Forward Chaining process which finds the candidate actions.

After ISDA handling finishes, ISDA dependent actions are checked. If an ISDA is found to be unusable, the ISDA dependent actions depending on this ISDA are also marked unusable by setting their isInvokable attributes to 0 in Action table. When the handling of composition problem is finished, isInvokable attributes of all ISDAs and ISDA dependent actions are set to the default value 1 again for the next problem. The pseudo code in Figure 4.4 illustrates the algorithm used for this process.

```
Process FindUnusableActions-Init()
       for each action in HasInitDependency
              dependencePredicate <- getDependencePredicate(action)
              # compare dependencePredicate with initial state predicates
              if not satisfies(ProblemInitialStatePredicates,
              dependencePredicate)
                     # update the action information in domain database
                     setIsInvokable(action, 0)
                     dependentActionList<-getISDADependentActions(action)
                     for each dependent in dependentActionList
                            setIsInvokable(dependent, 0)
                     end for
              end if
       end for
end Process
```

Figure 4.4: Pseudo code for Unusable Action Finding – Init

Unusable Action Finding – Goal:

In a similar way to the previous process, this process retrieves the goal state from the PDDL problem and checks the goal state dependent actions (GSDAs) and GSDA dependent actions which are stored in the pre-processing phase as described in Chapter 3. This process is an extension to the idea of pre-filtering described in [31] which does not include an analysis of services' problem goal state dependencies. When the user submits the problem to the system, all GSDAs and GSDA dependent actions are assumed to be usable for the submitted problem and their isInvokable attributes in Action table are set to 1 by default during the pre-processing phase which shows that the actions can be used for the given problem and they do not have any dependency to the problem goal state or their dependencies to the goal state are satisfied for this problem. This process firstly traverses the GSDAs in HasGoalDependency table in the domain database to check if at least one effect of each action in this table is a subpredicate of the predicates in the problem goal state. This check is done by comparing each effect of each action with the predicates in the goal state of the problem. If it is found that none of the effects of a GSDA satisfies at least one predicate in the problem goal state, isInvokable attribute of the GSDA in

Action table is set to 0 which shows that the action cannot be used to solve the given problem.

After handling GSDAs, GSDA dependent actions are checked. If a GSDA is found to be unusable, the GSDA dependent actions depending on this GSDA are also marked unusable by setting their isInvokable attributes to 0. When the handling of a composition problem is finished, isInvokable attributes of all GSDAs and GSDA dependent actions are set to default value 1 again for the next problem. The pseudo code in Figure 4.5 illustrates the algorithm used for this process.

```
Process FindUnusableActions-Goal()
       for each action in HasGoalDependency
              actionEffects <- getEffects(action)
              # check if the effects of action satisfies at least one
              # predicate in problem goal state
              if not satisfiesAtLeastOne(actionEffects,
              ProblemGoalStatePredicates)
                      # update the action information in domain database
                     setIsInvokable(action, 0)
                     dependentActionList<-getDependentActions(action)
                     for each dependent in dependentActionList
                            setIsInvokable(action, 0)
                     end for
              end if
       end for
end Process
```

Figure 4.5: Pseudo code for Unusable Action Finding – Goal

Forward Chaining:

This process makes use of a forward chaining algorithm similar to the one in [31] and selects the candidate actions that will be used to find the composite service for the given problem. This algorithm uses the chaining information which is stored in HasChain table during service preprocessing as described in Chapter 3. In addition, the forward chaining algorithm considers isInvokable values of actions during selection which are determined by the two unusable action finding processes described above. The forward chaining process runs concurrently with these two

processes and checks the information updated by them. The forward chaining algorithm uses two in-memory data structures called *action pool* and *predicate pool* which are empty at the beginning. The action pool stores the selected candidate actions. The predicate pool includes the predicates that are satisfied at an instant of the algorithm execution. In other words, at an execution instant, the predicate pool contains the union of the problem initial state predicates and the effects of candidate actions selected until that instant.

The algorithm runs as follows: First, it retrieves the predicates provided in the problem initial state and adds these predicates to the predicate pool. Then it executes a database query called initialQuery which checks the Action and HasPrec tables and finds the actions that can be invoked with the given initial state (i.e. the actions whose all preconditions are satisfied by the predicates in the initial state). The found actions are added to the action pool. After finding the candidate actions that can be invoked with the predicates in the problem initial state, the algorithm continues level by level. In each level a database query called levelQuery is executed. This query considers the effects of the candidate actions added to the action pool in the previous level and finds the actions whose at least one precondition can be chained with these effects. These precondition-effect chains are directly extracted from the information stored in HasChain table during service preprocessing. The resulting action set of levelQuery is called the possible actions. Afterwards, for each action in the possible actions, the predicate pool is checked to determine if all preconditions of the action can be chained with some predicates in the pool. In such a case, the possible action becomes a candidate action and it is added to the action pool. In addition, the effect predicates of the action are added to the predicate pool to be checked in the next level to find that level's possible actions.

While selecting the possible actions in each level, levelQuery also checks the value of isInvokable attribute for each action. This attribute is concurrently updated by the two unusable action finding processes. If isInvokable value of an action is 0, the action is discarded and is not considered as a possible action even if there is a predicate in the predicate pool that can be chained with a precondition of this action.

The algorithm ceases when all of the predicates in the problem goal state or their subpredicates become available in the predicate pool or when no more actions can be added to the action pool after finishing a level (i.e. when the size of the action pool remains same after two consecutive levels). The occurrence of the second condition means there does not exist any possible composite service for the problem given by the user because the actions in domain database cannot satisfy all of the predicates in the problem goal state. The pseudo code in Figure 4.6 illustrates the algorithm used for the forward chaining process.

```
Process ForwardChaining()
     # get initial and goal state from problem description
      initialPredicates <- getProblemInitialStatePredicates()
      goalPredicates <- getProblemGoalStatePredicates()

      # find the actions that can be invoked with predicates in
      # initial state and initialize predicatePool with initial state
      # predicates
      actionPool <- findInitiallyInvokableActions(initialPredicates)
      predicatePool <- initialPredicates

      # The effects that should be checked in first level are the
      # effects of the actions that can be invoked with initial
      # state predicates. These effects are then added to predicatePool
      checkPredicates <- getEffectPredicates(actionPool)
     addToPredicatePool(checkPredicates)

      # loop for each level
      while(true)
               # Find possible actions from checkPredicates
            possibleActions <- findPossibleActions(checkPredicates)
            delete(checkPredicates)

            for each possibleAction in possibleActions
                    # if all preconditions of action can be chained with
                    # a predicate in predicatePool, add to actionPool
                  if allPreconditionsCanBeChained(predicatePool,
                      possibleAction)
                        addToActionPool(possibleAction)
                        # add effects to predicates to be checked in
                        # next level and to predicatePool
                        addToCheckPredicates(getEffects(possibleAction
                        ))
                        addToPredicatePool(getEffects(possibleAction))
                  end if
            end for

            # If all predicates in goal state are satisfied, end process
            if areIncludedIn(goalPredicates, predicatePool)
                  return actionPool
            end if

            # If no actions added to actionPool, no composition exists
            if noActionsAddedToActionPool()
                  return NoCompositionExistsError
          end if
      end while
end Process
```

Figure 4.6: Pseudo code for Forward Chaining

In Figure 4.6, the predicatePool is stored as a hash table so that the search and update operations can be done quickly. The method allPreconditionsCanBeChained() considers the ontological relationships between

predicates while checking the predicate pool. In other words, the method returns true if for each effect predicate of the possible action, there is at least one predicate in the predicate pool which is a subpredicate of the effect. The methods findInitiallyInvokableActions() and findPossibleActions() execute initialQuery and levelQuery respectively. These queries are briefly explained as follows:

- **initialQuery.** Selects all actions from the domain database that satisfy both of these conditions: All preconditions of the action can be satisfied by the problem initial state predicates and the action is invokable.
- **levelQuery.** Selects all actions from the domain database that satisfy both of these conditions: At least one precondition of the action can be chained with the predicates in checkPredicates and the action is invokable.

Appendix A includes the actual SQL queries used in the implementation of initialQuery and levelQuery. The implementation of the algorithm in Figure 4.6 is done with Java programming language in a multi-threaded manner to allow effective utilization of hardware platforms with multiple processors. In such platforms, each created Java thread is assigned to one of the available processors by the Java Virtual Machine (JVM). In this algorithm, multiple threads are created at each level (i.e. the while loop in Figure 4.6) while checking whether the possible actions can be added to the action pool (i.e. the for loop in Figure 4.6). A new thread is created for each possible action in possibleActions list so that the checking process is done concurrently for all actions. If there exist sufficient number of processors, each one of these threads is assigned to a different processor, so allPreconditionsCanBeChained() method can be executed in parallel for all actions. This helps to decrease the time required for the ForwardChaining process and increase the overall performance of the Pre-Filtering component.

When the forward chaining process terminates, the execution of the Pre-Filtering component also terminates. If all of the predicates of the problem goal state are satisfied by the predicates in the predicate pool, the component returns a list of the

ids of the candidate actions in action pool which is retrieved from the Action table. This list is then delivered to the Composition/Execution Data Generator component. If the forward chaining algorithm returns NoCompositionExistsError, the Pre-Filtering component returns an error message to the user and terminates the whole system. The composition finding and service execution processes do not take place because during pre-filtering it is found that no composite service can be created to solve the problem provided by the user.

4.2.3 Generating Composition and Execution Data

This component retrieves the problem PDDL description and the ids of candidate actions if Pre-filtering component can find a set of candidate actions that satisfy the goal state in composition problem. This list of ids are used to generate the data required for finding a composite service that satisfies the user request and executing the services that constitute this composite service. The composition data consists of the problem PDDL description which is retrieved from the Pre-filtering component and the domain PDDL description of candidate services. The domain PDDL description is constructed as follows: First, for each action id, this component retrieves the parts of the corresponding action from the Action, HasPrec, HasEffect and Pred tables. The predicates used in the precondition and effect parts of the action are added to predicates section of the PDDL domain by creating dummy values for the predicate parameter names. After this step, the types used in these predicates are added to the types section of the domain PDDL. Types are added together with their supertypes that do not already exist in the types section. These supertypes are retrieved from HasSupertype table using the types. After adding predicates and types, the action itself is added to the PDDL domain. The parameters part of the action is created depending on the types used in preconditions and effects of the action. The dummy parameter names are created for each one of these types and these names are also used in the predicates in the precondition and effect parts of the action definition.

The service execution data consists of WSDL descriptions of the services that correspond to candidate PDDL actions and logical action-physical service mappings of the candidate actions which are stored during preprocessing phase. The service execution data is constructed as follows: First, the WSDL descriptions of candidate actions are extracted from the WSDL directory in the service repository and collected in another directory (WSDL Files in Figure 4.1). After preparing the WDSL descriptions, the logical action-physical service mapping of each candidate action is retrieved from the mappingXml attribute of the Action table in the domain database. These mappings are collected in a single xml file named OWLS-WSDLMapping.xml.

After creating the composition and execution data for candidate services, this component sends the PDDL domain description, PDDL problem description, WSDL files of candidate actions and OWLS-WSDLMapping.xml file to the existing WSCE framework to find the composite service and execute the services that constitute this composite service.

CHAPTER 5

INTEGRATION WITH THE WEB SERVICE COMPOSITION AND EXECUTION FRAMEWORK

The purpose of this thesis is to enhance the existing WSCE framework in [1] by providing scalability and maintenance of large service sets. The previous two chapters give the details of these enhancements: Chapter 3 explains the preprocessing system and the service repository that are used to store newly added service information and the details of how they help to maintain large service sets. In chapter 4, the details of the pre-filtering mechanism are presented and it is explained how this mechanism selects the candidate actions for a given problem and provides scalability. After detailing these enhancements, this chapter gives the details of integrating the preprocessing system and pre-filtering process to the existing WSCE framework.

The rest of this chapter is organized as follows: Section 5.1 explains the architecture of the preprocessing phase in [1] and Section 5.2 describes how this preprocessing phase is changed to be integrated with the pre-filtering process described in this thesis. Section 5.3 gives an overview of the interleaved composition and service execution phases of the existing WSCE framework. Finally, section 5.4 explains the unexpected event handling phase in [1] and how the information provided after this phase is used to remove the unreachable services from the service repository.

5.1 Preprocessing Phase of the Web Service Composition and Execution Framework

The preprocessing phase in [1] deals with the inputs from the service providers and users. By using these inputs, this phase prepares the PDDL data for Simplanner to find the composition and creates the service execution environment which is used for real service execution. Figure 5.1 which is adapted from [1] shows the architecture of the preprocessing phase.

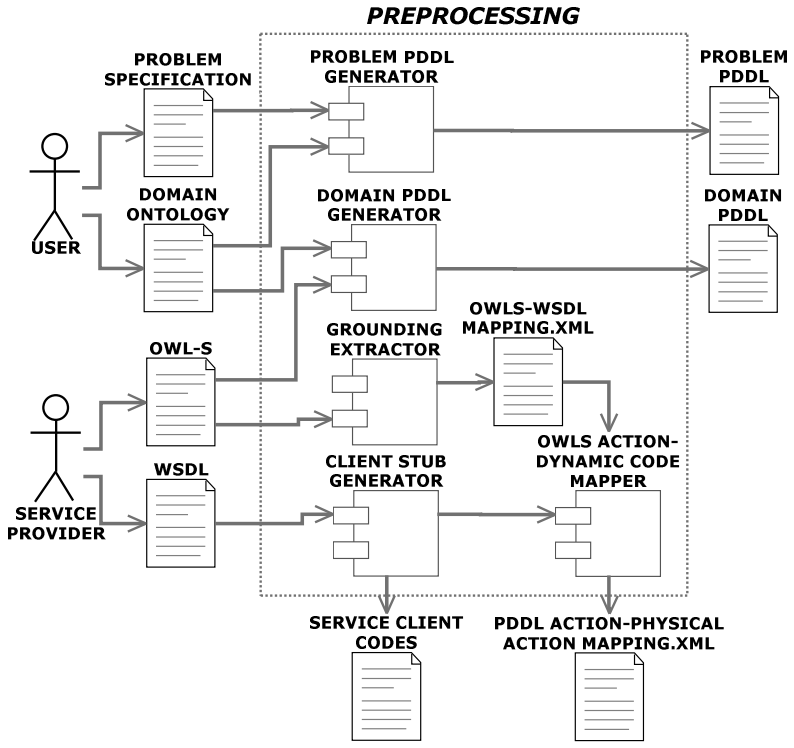

Figure 5.1: Preprocessing component of WSCE framework

In this phase, the user provides the composition problem in OWL and the service provider provides OWL-S and WSDL description of the services. Since [1] does not use any efficient service storage and pre-filtering mechanism to store the available

services and select the ones relevant to the user problem, the provided service descriptions include all available services.

The provided problem and service descriptions are used by the problem PDDL generator and the domain PDDL generator components to create the problem and domain PDDL descriptions for the planning phase. Since the domain PDDL generation is done together with the problem PDDL generation in preprocessing phase, this brings an overhead to the performance of the framework because the same service descriptions are converted to PDDL domain repeatedly for each composition problem. The service descriptions are also used by the grounding extractor component to create mappings for the WSDL descriptions of the services. This step is also repeated for each new problem even if the service set remains the same. The resulting mappings created by the grounding extractor component are provided to the OWL-S Action-Dynamic code mapper component together with the dynamically created client information retrieved from the client stub generator component. The OWL-S Action-Dynamic code mapper component uses this information to add client code information to the provided mappings.

5.2 Pre-filtering Integration to WSCE Framework

The purpose of the composition and execution data creator system described in Chapter 4 is to provide the information required by the existing WSCE framework. With the help of this system, some of the information created by the preprocessing phase in Figure 5.1 is directly provided to other phases of the WSCE framework. Since the service information stored in domain database is in PDDL format in this thesis, the problem PDDL and domain PDDL generator components in Figure 5.1 are not required and the problem PDDL and domain PDDL descriptions are directly provided by the composition and execution data generator component to the planning phase in [1]. Furthermore, as OWLS-WSDL mapping information for each service is stored in the domain database, the grounding extractor component in Figure 5.1 is not required either. The OWLS-WSDL mapping.xml file is directly

created by the composition and execution data generator component described in Chapter 4.

The removal of these components from the preprocessing phase in [1] also helps to decrease the time required to solve a composition problem because most of the service data processing done in this phase is moved to other components of the system and this processing is not repeated for every given composition problem. The conversion of each service description to PDDL is done only once during the preprocessing phase described in chapter 3 and this PDDL information is stored in the domain database. Therefore no PDDL conversion is required for services after the user gives the composition problem. Furthermore, since grounding extraction is also done while storing the service information to the domain database, this grounding information is not searched from OWL-S description while dealing with a composition problem. Figure 5.2 illustrates the integration of composition and execution data creator system to the preprocessing phase of [1] in Figure 5.1.

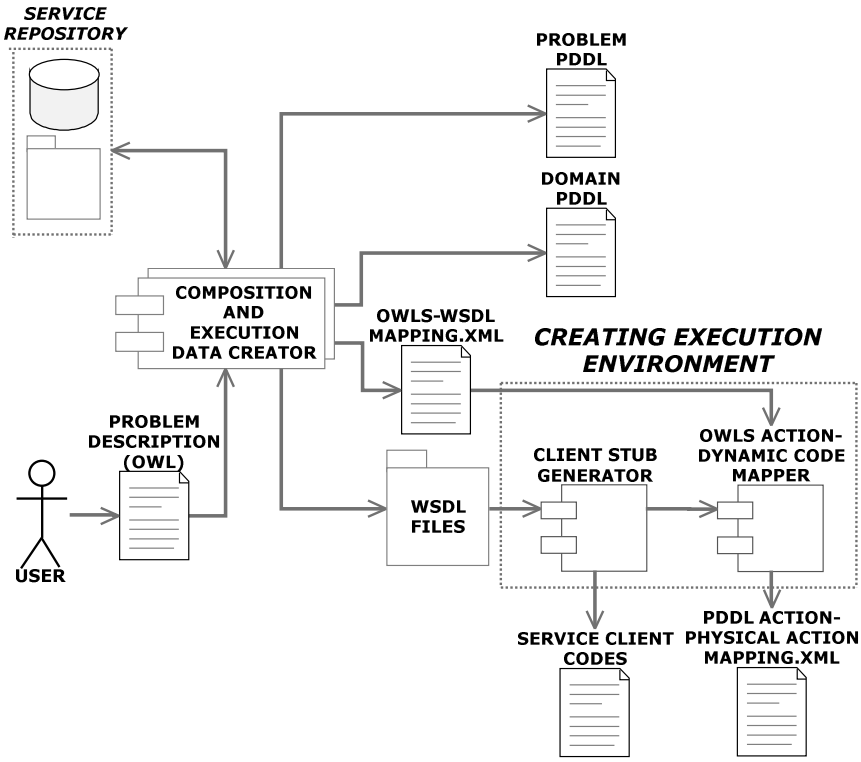

Figure 5.2: Integration of composition and execution data creator with preprocessing phase in WSCE Framework

With the integration illustrated in Figure 5.2, the preprocessing phase in figure 5.1 is replaced with an execution environment creation phase. This new phase retrieves the service WSDL descriptions from the composition and execution data creator system together with the created mappings and it creates the service client codes and updated mappings which include the created client information.

The new integrated system in Figure 5.2 provides the same types of information with the preprocessing phase of the existing WSCE framework. This information consists of the problem PDDL, domain PDDL, service client codes and PDDL Action-Physical Action Mapping.xml. The difference of the new integrated system is that these four types of information are constructed based on the candidate actions found by the pre-filtering component of the composition and execution data creator system. On the other hand, the preprocessing phase in [1] uses all actions available in the

domain and creates these four types of information for all actions in the domain. This is the main contribution of this thesis that provides scalability to the existing WSCE framework. When the number of actions in domain is large, the created PDDL domain by preprocessing phase in [1] also becomes large. This prevents Simplanner to return timely responses during composition. However, since the pre-filtering component in this thesis selects only the actions that can be involved in the composite service for the given problem, the domain PDDL is created only from these candidate actions and Simplanner uses only this small domain to find the composition.

5.3 Interleaved Web Service Composition and Execution

After the integrated system in Figure 5.2 creates the PDDL descriptions and the service execution data, this information is used by the planning and execution phases of the existing WSCE framework. In the planning phase, all required work is handled by Simplanner [6]. Initially, Simplanner uses the available PDDL problem and domain descriptions and produces the possible logical action instances via grounding. After grounding, Simplanner creates an initial plan and continues to plan during the lifetime of the session. Being an any-time planner, Simplanner produces a logical action in a short time and continues planning as time permits. The produced action is delivered to the service executor for real execution. While the real service execution takes place, Simplanner continues to construct and refine the current plan. This property of Simplanner enables interleaving planning and service execution phases. The existing WSCE framework also handles the problems that may appear during service execution such as information unavailability and service execution failures. Such cases are handled by the unexpected event handler of the WSCE framework which examines the current state and informs Simplanner about the problem. Since Simplanner is able to run on dynamic plan states, it produces a new plan by considering the occurred problem and the current plan state. Figure 5.3 below shows the interleaved composition and execution phases of the existing WSCE framework.

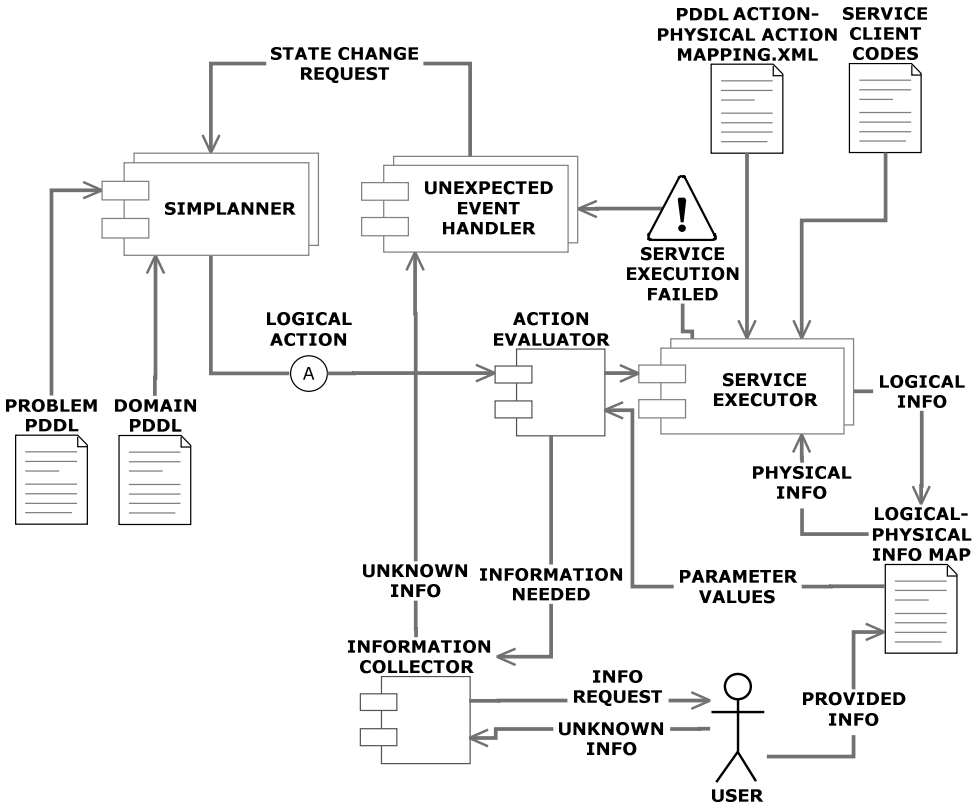

Figure 5.3: Interleaved composition finding and service execution in WSCE framework

After a logical action is prepared by Simplanner for real execution, the parameters of the action are examined by the action evaluator and their real values are retrieved from logical-physical information mapping. If the information is unknown in the mapping, the user is asked for the unknown information. If the user provides this information, execution continues. Otherwise, the unexpected event handler is notified to deal with the problem.

The service execution is done by the service executor in two ways according to the service behavior. If the service being executed is an information gathering service, it is done as a usual service call. On the other hand, if a world altering service is being executed, the service call conforms to WS-Business Activity and WS-Coordination specifications. During execution, the service client codes and the PDDL action-

Physical action mappings which are produced by the execution environment creator in Figure 5.2 are used. The real values of service call arguments are collected from the logical/physical information map. If the service provides information, the provided information is stored in the logical/physical information map and the consumed information is removed. If the service has world altering effects, its call is done through the business activity coordinator which is generated at the beginning of each session.

5.4 Unexpected Event Handling and Service Repository Integration

At run time, the service execution or information collection may fail unexpectedly because of reasons like network problems, service unavailability, wrongly provided arguments or the user may not know the information that is required by the service. Such cases are handled in the unexpected event handling phase of the existing WSCE framework. In this phase, initially it is assumed that all specified services can be executed and all the information required by services can be provided by the user. If a service execution fails, the corresponding service is made logically unavailable. As a result, the logical action that corresponds to the failed service is not considered by Simplanner at later steps. Simplanner tries to find alternative actions to achieve the user goal. If alternative actions do not exist, no plans can be found to satisfy the user goal.

When a web service requires some input information, if the required information cannot be provided by the user, the logical state that assumes all the required information can be provided by the user at runtime is modified. The logical counterpart of the information which is not known by the user is removed from the planning state. As a result, Simplanner tries to find an action that provides the required information or an alternative path that does not require that particular information. If the planner finds such an action, the execution continues; otherwise the session is terminated. If re-planning cannot create new solutions to achieve the user goals after some unexpected events, the transactional operations are rolled back

to prevent the side effects of the unsuccessful execution attempts. Figure 5.4 shows the unexpected event handling phase in [1] and the service repository integration (bold arrows and grey component in the figure) added to this phase in the scope of this thesis.

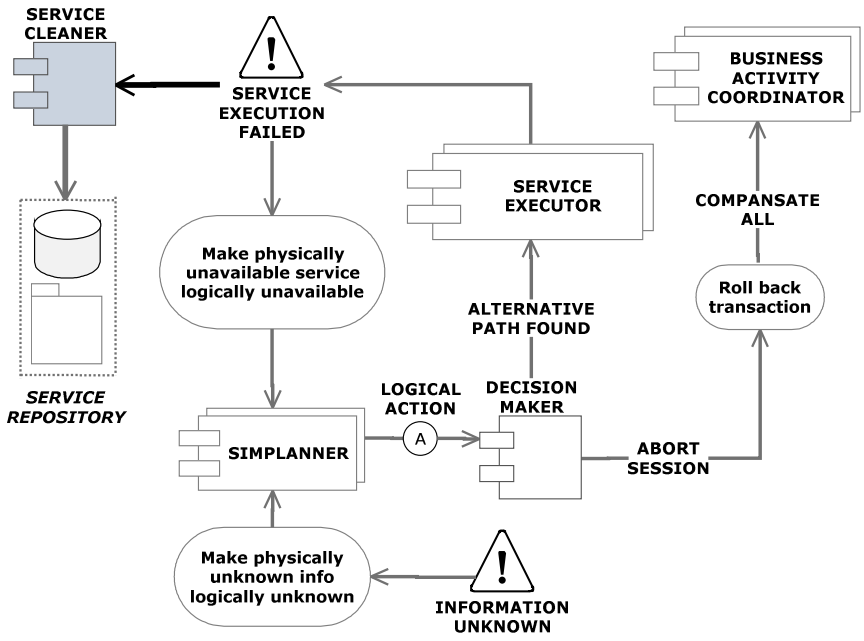

Figure 5.4: Unexpected event handling and service repository integration

When it is observed that a service is unavailable, the information stored in the service repository for that service is cleaned. This is done by the service cleaner component as follows: When the service cleaner component retrieves the name of the failed service, it first finds the action id, effect and precondition ids from Action, HasEffect and HasPrec tables in the domain database for the corresponding action. Using the effect and precondition ids, the service cleaner removes the chains from HasChain table. While cleaning the chains, the actions chained with the failed action are checked to see if the removal of the chain creates initial or goal state dependency for these actions. If a dependency occurs for an action, this action is added to

HasInitDependency, HasGoalDepency, HasActionDependencyInit, HasActionDependencyGoal according to the type of the occurred dependency. After adding the occurred dependencies for the actions that have removed chains with the failed action, these four tables are checked for the failed action. The rows that include the failed action are removed from the tables. When the chain clean-up of the failed action is finished, the action information is removed from HasEffect, HasPrec and Action tables by using the ids of the action, its effects and preconditions. Finally, the failed service is searched in the WSDL directory and the corresponding WSDL description is removed from this directory.

CHAPTER 6

EXPERIMENTAL EVALUATION

This chapter explains the details of the experimental evaluation done in this thesis. The performance of the pre-filtering mechanism that finds the candidate actions for a composition problem should be reasonable to be able to provide scalability to the existing WSCE framework. The purpose of the conducted experiments is to observe the performance of the pre-filtering mechanism on data sets involving large domain ontologies and large number of services. For this purpose, five different test sets were executed which consist of a varying number of concepts in domain ontologies and a varying number of services.

The rest of the chapter is organized as follows: First section describes the content of the test data used in the experimental evaluation. Section 2 explains the platform used for the experimental evaluation and Section 3 shows the results of the experiments.

6.1 Experimental Data

In this thesis, the experiments are done with the test dataset of Web Services Challenge'09 (WS-Challenge'09) [36]. In WS-Challenge'09 the data formats and the contest data is based on OWL, WSDL and WS-BPEL schemas for ontologies, services and service orchestrations. Furthermore, the annotation of each service includes quality of service (QoS) information in terms of response time and throughput of the service. The data set consist of five different test sets each of

which includes service descriptions and a composition problem description based on a domain ontology. The task in the challenge is to find a composition of services that produces a set of queried output parameters from a set of given input parameters and a set of service descriptions. In addition, the found composition should be the one with the least response time and the highest possible throughput.

The composition problem and the service descriptions in the test sets of WS-Challenge'09 are adapted to the system in this thesis as follows: Since the non-functional properties of services are out of the scope of this thesis, QoS information available in the service descriptions is not used. Moreover, in this thesis, no WS-BPEL description is created after finding the composition. Therefore, WS-BPEL specifications in the test sets are also ignored. The rest of this section describes the other contents of the information in the test sets and how they are used for the evaluation of the system in this thesis.

6.1.1 Domain Ontology

In WS-Challenge'09, the domain ontology is expressed with OWL. The domain is strictly limited to taxonomies consisting of sub and super class relationship between semantic concepts. In addition to class definitions, domain ontology includes definitions of some individuals which are instances of the classes. These individuals are used to annotate input and output parameters of the services. Since the pre-filtering mechanism in this thesis uses PDDL data, OWL class definitions in the test sets are converted to PDDL type hierarchy. Furthermore, the individual definitions are parsed and stored in an in-memory data structure to be used in finding the input-output types of the services.

6.1.2 Service Descriptions

The service descriptions in WS-Challenge'09 test sets are prepared with an extended version of WSDL which includes semantic annotation fields as well as syntactic descriptions. The semantic annotation of services is done with Mediation Contract Extension (MECE) [37] which is a simpler annotation format when compared with

OWL-S. In addition to extended WSDL descriptions, test sets include a much simpler XML description of services which shows the inputs and outputs of services in terms of the individuals defined in the domain ontology. Figure 6.1 illustrates an example service description with this XML format.

```
<service name="serv1859188453">
        <inputs>
            <instance name="inst1078434457"/>
            <instance name="inst817090775"/>
        </inputs>
        <outputs>
            <instance name="inst1979825120"/>
        </outputs>
</service>
```

Figure 6.1: Example service description from WS-Challenge'09 test sets

In the test sets, semantic annotation is only used to describe the ontological relationships of the input-output parameters of the services. The services in the test sets do not include any precondition or effect description and also no OWL-S annotation is available. Therefore, the OWL-S to PDDL conversion rules defined in Chapter 3 cannot be applied to these test sets. The PDDL action descriptions are created simply by creating agentHasKnowledgeAbout(x) predicates for the inputs and outputs of the services. To illustrate, if we assume that the concept definitions in the domain ontology that correspond to individuals inst1078434457, inst817090775, inst1979825120 are con1, con2 and con3 respectively, the PDDL action counterpart of the service in Figure 6.1 is as illustrated in Figure 6.2.

```
(:action serv1859188453

:parameters ( ?inst1078434457-con1 ?inst817090775–con2 ?inst1979825120–con3)

:precondition (and (agentHasKnowledgeAbout ?inst1078434457)
(agentHasKnowledgeAbout ?inst817090775))

:effect (and (agentHasKnowledgeAbout ?inst1979825120)
))
```

Figure 6.2: PDDL action that corresponds to the service description in figure 6.1

Since the services do not include any precondition or effect description, the only available predicate in the PDDL domain is agentHasKnowledgeAbout(x) which is used for input-output description of services. Therefore, the predicates part of the PDDL domain descriptions consists of only this predicate.

6.1.3 Problem Description

In WS-Challenge'09 test sets, the composition problem is presented in an XML format as illustrated in Figure 6.3. The individuals inside 'provided' part of the descriptions are the instances of the concepts available in problem initial state. The individuals inside 'wanted' part are the instances of the concepts required in goal state. Figure 6.4 shows the PDDL counterpart of this problem description with the assumption that the concepts that correspond to inst30807040, inst1625495672 and inst1315200283 in the domain ontology are con1, con2 and con3 respectively.

```
<task>
        <provided>
                <instance name="inst30807040"/>
        </provided>
        <wanted>
                <instance name="inst1625495672"/>
                <instance name="inst1315200283"/>
        </wanted>
</task>
```

Figure 6.3: Example problem description from WS-Challenge'09 test sets

```
(define (problem testsetXXproblem)

(:domain testsetXX)

(:objects inst30807040 - con1 inst1625495672 - con2 inst1315200283- con3 )

(:init (agentHasKnowledgeAbout inst30807040)
)

(:goal (and (agentHasKnowledgeAbout inst1625495672)
(agentHasKnowledgeAbout inst1315200283)
))

)
```

Figure 6.4: PDDL problem that corresponds to the problem description in figure 6.3

After converting the domain ontology, service descriptions and problem description of a test set to PDDL domain and problem descriptions, these descriptions are used to perform tests to evaluate the performance of pre-filtering mechanism described in Chapter 4. Table 6.1 illustrates the total service and concept numbers together with the concept numbers in the initial and goal states of the composition problem for each test set.

Table 6.1: Number of services and concepts in test sets of WS-Challenge'09

Test set	Number of services	Number of total concepts	Number of concepts in initial state	Number of concepts in goal state
Testset01	572	1578	10	4
Testset02	4129	12388	9	3
Testset03	8138	18573	1	4
Testset04	8301	18673	7	3
Testset05	15211	31044	9	2

6.2 Experimental Environment

Since the pre-filtering mechanism described in Chapter 4 is designed to utilize hardware platforms involving multiple CPUs, we performed our tests on High Performance Computing (HPC) System of METU Computer Engineering Department [38]. The hardware and software properties of this system that are relevant to our tests are as follows:

- **Hardware Properties:** The system consists of 46 CPUs for processing, each of which consists of 4 cores. There is also 46x16 GB RAM storage which makes 736 GB memory. The total disk storage in the system is 6.5 TB.
- **Software Properties:** The operating system running on HPC system is Scientific Linux 4.5 64-bit which is an open source Linux distribution derived from Red Hat Enterprise Linux [42]. The used file system is Lustre which is a parallel disk file system generally used for large scale cluster computing [43]. The Java platform on the system consists of Java SE Runtime Environment 1.6.0_20-b02 and javac 1.6.0_20.

6.3 Experiment Results

Before conducting the experiments, first the service data in test sets are converted to PDDL action descriptions as described in Section 6.1. After this conversion, preprocessing phase described in Chapter 3 is executed on the PDDL data and service composition data (i.e. PDDL types, predicates, actions and the chains and dependencies of actions) is stored in service repository.

The experiments start by converting the problem description to PDDL. Then the pre-filtering system described in chapter 4 runs with the test set information stored in service repository during preprocessing phase. Two types of experiments were done which depend on the termination condition of pre-filtering algorithm:

- **Termination right after satisfying the goals:** In this type of experiment, the pre-filtering algorithm terminates right after it finds the minimum set of candidate actions that satisfy the goals of the composition problem. This means that after termination of the algorithm, the selected candidate actions in the action pool include at least one composition that satisfies the problem goal but not necessarily all possible compositions.
- **Termination when no actions can be added to action pool:** In this type of experiment, the pre-filtering algorithm continues to run until it is found that the size of the action pool remains same after two consecutive iterations. In this case no more actions can be added to the action pool and the set of candidate actions remains the same. This means that the selected candidate actions in the action pool include all possible compositions that satisfy the problem goal.

Table 6.2 shows the results of the two types of the tests done for each test set in Table 6.1. For each type, the first row shows the size of the action pool (i.e. the number of selected candidate actions from the whole action set) after the pre-

filtering algorithm terminates. The second row shows the time passed for the corresponding test.

Table 6.2: Experiment results

Termination Condition	Testset01	Testset02	Testset03	Testset04	Testset05
Single Solution	47	91	63	197	214
	12 ms	22 ms	19 ms	38 ms	46 ms
All Solutions	79	139	152	329	236
	19 ms	28 ms	33 ms	50 ms	49 ms

As it can be inferred from the Table 6.1 and 6.2, the pre-filtering algorithm provides an important decrease in the number of actions in the domain and the time required for this process is reasonable. In addition, the experiment results in Table 6.2 show that the time required for pre-filtering increases proportionally with the size of the concepts and actions in the domain. As expected, the tests that find all possible solutions for the given problem create larger sets of candidate actions and pre-filtering process takes more time when compared with the tests that terminate right after finding a single solution.

In order to evaluate the increase in the performance of Simplanner with the filtered domains, the service execution step in interleaved composition and execution phase is disabled and the time required for Simplanner to achieve all the goals in the problem description is calculated with three different experiments. First experiment is done for the case that pre-filtering terminates right after satisfying the tha goals. Second experiment covers the case where pre-filtering algorithm terminates when no more actions can be added to action pool. Finally, third experiment is done for the case that no pre-filtering algorithm is applied to the action domain. Table 6.3 shows

the time required by Simplanner to achieve all the problem goals with Testset01 in Table 6.1 for the three different pre-filtering levels.

Table 6.3: Simplanner performance with Testset01 for different pre-filtering levels

Pre-filtering level	# of actions and types	Time to achieve all goals (seconds)
Termination after finding single solution	47 466	132
Termination after finding all solutions	79 466	263
No pre-filtering	572 1578	No response.

As Table 6.3 illustrates, the time required by Simplanner to achieve all the goals of the problem increases with the size of domain. In the experiment where no pre-filtering is applied, Simplanner returned out of memory error and could not run to find a plan. As this experiment illustrates, in addition to significantly decreasing the time required for planning, the pre-filtering algorithm helps Simplanner to run successfully in the cases that it cannot run with non-filtered domains.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In today's web technology, web services and service oriented architectures (SOAs) have a quite important role in achieving an effective interoperability among different businesses. This is achieved with the platform independent standards used in web service implementations. With this emerging role of web services and SOAs, the requirement of automatically discovering, composing and executing web services has begun to draw high attention, and today the approaches proposed to find solutions to these automation problems constitute a hot research area of computer science. Finding effective solutions to these problems is very important for the future of web services because as the number of available services in the web increases, the manual discovery of suitable web services from a huge service set and composing and executing these services to provide a required functionality becomes impractical.

Most of the approaches that address these problems benefit from semantic web and semantic web service technologies. These technologies enable the representation of semantic information of services and this information facilitates the automated discovery and composition of services. The existing WSCE framework described in [1] also makes use of semantic web services and provides interleaved composition and execution of web services via a domain independent AI planner. This framework proposes solutions for many open problems defined in the web service composition literature. However, it does not provide solutions to two important problems in WSC approaches: scalability and service domain maintenance. Scalability is a common problem in the approaches using domain independent AI

planners. These approaches fail to scale well with the increasing number of services. Since there are a huge number of services in real world scenarios, scalability has to be achieved to provide practical significance for the proposed approach. In addition, the existing WSCE framework does not enable storing and maintaining service information in a service repository. It simply considers the services in the domain as a single OWL-S description file provided by the user at the beginning of the composition request. This approach is not practical because most of the time, the user does not know the available services, so the service descriptions cannot be provided by the user to the system. In addition, when the services in the whole web are considered, collecting the service descriptions in a single file is practically impossible.

This thesis proposes an extended WSCE approach that enhances the existing WSCE framework by providing scalability and service domain maintenance. These enhancements are provided as follows:

- **Service Domain Maintenance:** This thesis adds a service repository and a pre-processing system to the existing WSCE framework which enables storing service information permanently in the system. When a service provider wants to add service information to service repository, this information is processed by pre-processing system and service information required for automated service composition and execution is added to service repository. In addition, the pre-processing phase extracts some additional information from service descriptions, namely chaining information and problem state dependencies, and stores this information in service repository as well. This additional information is used during problem handling to decrease the time required to execute the pre-filtering algorithm which finds the candidate services for the composition.
- **Scalability:** This thesis provides scalability to the existing WSCE framework by adding a pre-filtering phase to the framework. When the user sends the composition problem to the framework, the pre-filtering phase executes a

forward chaining algorithm and selects the candidate services from the service repository that can be used in the composition. These selected services are then sent to the AI planner to find the composite service and execute the services that are used in the composite service. The conducted experiments show that in a reasonable time, pre-filtering phase effectively reduces the size of service set that is provided to AI planner for the composition. This helps to prevent the exponential enlargement of planning search space and AI planner finds the composition quickly by considering only the reduced candidate service set.

In addition to these enhancements, this thesis explains how the added components are integrated to the existing WSCE framework.

This thesis provides two important enhancements to the existing WSCE framework. However some improvements still exist as future work. The most important future work is to provide a parallel execution of pre-filtering and planning phases. In the current framework, these two phases run sequentially. First, the pre-filtering phase runs and selects the candidate services for the composition. Afterwards, Simplanner runs and finds the composite service. If these two phases run in parallel, the services found in each iteration of the pre-filtering phase can be dynamically provided to Simplanner and Simplanner can use this dynamic set of services while building the composite service. This functionality can be achieved by enhancing Simplanner with the ability to run with a dynamic action set. Simplanner is able to run with dynamic planning states but this property is not sufficient to provide parallel filtering and planning.

Another future work is to consider services' Quality of Service (QoS) parameters during the pre-filtering phase. The QoS can be considered in the cases when more than one service has similar IOPE values and these services are selected during pre-filtering. In such a case, the service that provides the highest QoS can be selected and the other services that have similar IOPE values can be discarded from the composition.

Lastly, a future work can be to move the domain database in the service repository from an in-memory database to a disk database and use an in-memory database as a caching mechanism. This change will help to store huge service sets in the domain database in the environments that do not have sufficient in-memory space while keeping the performance of in-memory database querying and updating. To achieve maximum caching performance, the required changes to the current database design should be analyzed.

REFERENCES

[1] Kuzu, M. and N.K. Cicekli, Dynamic Planning Approach to Automated Web Service Composition, accepted for publication in Applied Intelligence, 2010.

[2] Christensen E., Curbera F., Meredith G., Weerawarana S, "Web Services Description Language (WSDL) 1.1", http://www.w3.org/TR/wsdl, last visited on 20.05.2010.

[3] Smith M.K., Welty C., McGuinness D. L., "OWL Web Ontology Language Guide", http://www.w3.org/TR/owl-guide, last visited on 20.05.2010.

[4] Martin D., Burstein M., Hobbs J., Lassila O., McDermott D., McIlraith D., Narayanan S., Paolucci M., Parsia B., Payne T., Sirin E., Srinivasan N., Sycara K., "OWL-S: Semantic Markup for Web Services", http://www.w3.org/Submission/OWL-S, last visited on 20.05.2010.

[5] Ghallab M., Howe A., Knoblock C., McDermott D., Ram A., Veloso M., Weld D., Wilkins D., "PDDL: The Planning Domain Definition Language, AIPS-98 Planning Committee, 1998.

[6] Sapena O., Onaindia E., "Planning in Highy Dynamic Environments: An Anytime Approach for Planning Under Time Constraints", Journal of Applied Intelligence, Volume 29, Number 1, pages 90-109, 2007.

[7] OASIS Web Services Business Activity Specification, http://docs.oasis-open.org/ws-tx/wsba/2006/06, last visited on 20.05.2010.

[8] Milanovic N., Malek M., "Current Solutions for Web Service Composition", IEEE Transactions on Internet Computing, Volume: 8, Issue: 6, pages 51-59, 2004.

[9] Srivastava B., Koehler J., "Web Service Composition – Current Solutions and Open Problems", ICAPS 2003 Workshop on Planning for Web Services, 2003.

[10] Polleres A, "AI Planning For Web Service Composition", Presentation, Ilog, Paris, France, 2004. http://axel.deri.ie/~axepol/presentations/20040907-paris-ilog-AIplanning4WSC.ppt, last visited on 20.05.2010.

[11] Agarwal V., Chafle G., Mittal S., Srivastava B., "Understanding Approaches for Web Service Composition and Execution", IBM Research Report, 2007.

[12] Rao J., Su X., "A Survey of Automated Web Service Composition Methods", Proceedings of 1sth International Workshop on Semantic Web Services and Web Process Composition, pages 43-54, 2004.

[13] Yu L., "Introduction to Semantic Web and Semantic Web Services", CRC Press (Boca Raton, FL), 2007.

[14] Haas H., Brown A., "Web Services Glossary", http://www.w3.org/TR/ws-gloss/, last visited on 20.05.2010.

[15] Box D., Ehnebuske D., Kakivaya G., Layman A., Mendelsohn N., Nielsen H. F., Thatte S., Winer D., "Simple Object Access Protocol (SOAP) 1.1", http://www.w3.org/TR/2000/NOTE-SOAP-20000508/, last visited on 20.05.2010.

[16] Bellwood T., "UDDI Version 2.04 API Specification", http://www.uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm, last visited on 20.05.2010.

[17] Oh S., Lee J., Cheong S., Lim S., Kim M., Lee S., Park J., Noh S., Sohn M., "WSPR*: Web-Service Planner Augmented with A* Algorithm", IEEE Conference on Commerce and Enterprise Computing, pp.515-518, 2009.

[18] Le D. N., Goh A. E., Cao T. H., "A survey of web service discovery systems" International Journal of Information Technology and Web Engineering, vol. 2, pp. 65-80, 2007.

[19] Russel S., Norvig P., "Artificial Intelligence: A Modern Approach", 3rd edition, 2003.

[20] Blum A., Furst M., "Fast Planning Through Planning Graph Analysis", Proceedings of 14th International Joint Conference on Artificial Intelligence, pp. 1636-1642, 1995.

[21] STRIPS language, http://en.wikipedia.org/wiki/STRIPS, last visited on 20.05.2010.

[22] ADL language, http://en.wikipedia.org/wiki/Action_description_language, last visited on 21.07.2009.

[23] Helmert M., "An Introduction To PDDL", http://www.cs.toronto.edu/~sheila/2542/w09/A1/introtopddl2.pdf, last visited on 20.05.2010.

[24] Garofalakis J., Panagis Y., Sakkopoulos E., Tsakalidis A., "Web Service Discovery Mechanisms: Looking for a Needle in a Haystack?", International Workshop on Web Engineering, in conjunction with ACM Hypertext, 2004.

[25] Casati F., Ilnicki S., Jin L., "Adaptive and Dynamic Service Composition in EFlow", Proceedings of 12th International Conference on Advanced Information Systems Engineering, 2000.

[26] Chan M., Bishop J., Baresi L., Survey and Comparison of Planning Techniques for Web Services Composition, University of Pretoria Pretoria, Technical Report, 2007.

[27] In-memory Database, http://en.wikipedia.org/wiki/In_memory_database, last visited on 20.05.2010.

[28] Garcia-Molina H., Salem K., "Main Memory Database Systems: An Overview," IEEE Transactions on Knowledge and Data Engineering, vol. 4, no. 6, pp. 509-516, 1992.

[29] Klusch M., Gerber A., Schmidt M., "Semantic Web Service Composition Planning with OWLS-XPlan", Proceedings of the AAAI Fall Symposium on Semantic Web and Agents, AAAI Press, 2005.

[30] "Extreme Performance Using Oracle TimesTen In-Memory Database", An Oracle White Paper, last visited on 20.05.2010.

[31] Bartalos P., Bielikova M., "Semantic Web Service Composition Framework Based on Parallel Processing", 2009 IEEE Conference on Commerce and Enterprise Computing, pp. 495-498, 2009.

[32] Akkiraju R., Srivastava B., Ivan A., Goodwin R., Syeda-Mahmood T., "SEMAPLAN: Combining Planning with Semantic Matching to Achieve Web Service Composition", IEEE International Conference on Web Services (ICWS'06), pp. 37-44, 2006.

[33] Yan Y., Zheng X., "A Planning Graph Based Algorithm for Semantic Web Service Composition", 2008 10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services, 2008.

[34] Kwon J., Hyeonji K., Lee D., Lee S., "Redundant-Free Web Services Composition Based on a Two-Phase Algorithm" , 2008 IEEE International Conference on Web Services, E-Commerce and E-Services, pp.361-368, 2008.

[35] PDDL4J, http://sourceforge.net/projects/pdd4j, last visited on 20.05.2010.

[36] The Web Service Challenge Rules, http://ws-challenge.georgetown.edu/wsc09/downloads/WSC2009Rules-1.1.pdf, last visited on 20.05.2010.

[37] Bleul S., Comes D., Zapf M. "Self-Integration of Web Services in BPEL Processes", Proceedings of SAKS Workshop, Verlag: University of Kassel, 2008.

[38] METU CENG High Performance Computing, http://hpc.ceng.metu.edu.tr, last visited on 20.05.2010.

[39] Kim H., Kim I., "Mapping Semantic Web Service Descriptions to Planning Domain Knowledge", Proceedings of IFMBE, Volume 14, pages 388-391 Springer Berlin Heidelberg, 2007.

[40] OWLS2PDDL tool, http://projects.semwebcentral.org/projects/owls2pddl/, last visited on 20.05.2010.

[41] Axis, Apache Web Services Project, http://ws.apache.org/axis/, last visited on 20.05.2010.

[42] Scientific Linux, https://www.scientificlinux.org, last visited on 20.05.2010.

[43] The Lustre File System, http://wiki.lustre.org/index.php/Main_Page, last visited on 20.05.2010.

[44] Peer J., "Semantic Service Markup with SESMA", Proceedings of International World Wide Web Conference, 2005.

[45] Peer J., "Web Service Composition as AI Planning – a Survey", Technical report, Univ. of St. Gallen, 2005.

[46] Digiampetri L., Alcazar J., Medeiros C. B., "AI Planning in Web Services Composition: a review of current approaches and a new solution", 2007.

[47] Kona S., Bansal A., Gupta G., "Automatic Composition of Semantic Web Services", Proc. of IEEE International Conference on Web Services (ICWS), 2007.

[48] Bartalos P., Bieliková M., "Fast and Scalable Semantic Web Service Composition Approach Considering Complex Pre/Postconditions," services, pp.414-421, Congress on Services - I, 2009.

[49] Nebel B., Dimopoulos Y., Koehler J., "Ignoring irrelevant facts and operators in plan generation.", In: Proc. ECP-97, Toulouse, France, 1997.

APPENDIX A

PRE-FILTERING QUERIES

initialQuery:

```
SELECT  aid
FROM        Action
WHERE   isinvokable = 1 AND
          aid   NOT IN (
    SELECT  aid
    FROM        HasPrec
    WHERE   predid  NOT IN (
        SELECT  superpid
        FROM        HasSuperpred
        WHERE   pid IN (
            -- get the pids of predicates provided in
            -- initial state of the problem
            SELECT  pid
            FROM        Pred
            WHERE   predicate = 'initPred1' OR
                    Predicate = 'initPred2' OR
                    Predicate = 'initPred3'
        )
    )
);
```

levelQuery:

```
SELECT  act.aid, pred.predicate, prec.predid
FROM        Action act, HasPrec prec, Pred pred
WHERE   prec.predid = pred.pid      AND
        act.aid = prec.aid      AND
        act.isinvokable = 1     AND
        prec.aid    IN (
    SELECT  aid
    FROM        HasPrec
    WHERE   pid IN (
        SELECT  pid
        FROM        HasChain
        WHERE   eid IN (
            SELECT  eid
            FROM        HasEffect
            WHERE   predid  IN (
                -- get the eids of predicates to be
                -- checked in this level
                SELECT  pid
                FROM        Pred
                WHERE   predicate='checkEffect1'    OR
                        predicate='checkEffect2'    OR
                        predicate='checkEffect3'
            )
        )
    )
);
```