Automating Service Composition and Delivery in Highly Heterogenous Distributed Environments

by

Hossein Pourreza

A Thesis submitted to the Faculty of Graduate Studies of

The University of Manitoba

in partial fulfilment of the requirements of the degree of

Doctor of Philosophy

Department of Computer Science

University of Manitoba

Winnipeg, Manitoba, Canada

Automating Service Composition and Delivery in Highly Heterogenous Distributed Environments

By

Hossein Pourreza

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

Of

Doctor of Philosophy

Abstract

In a distributed environment, devices and components can share their resources by offering different services. By abstracting devices using the services that they offer, we can define a service-based architecture that readily supports the discovery and creation of new, composite services to expand the capabilities available in the distributed environment. Pervasive environments such as a Home Area Network (HAN), meeting/conference room, etc. are examples of a highly heterogeneous distributed environment that can exploit such a service-based architecture. In this thesis, I propose an architecture for service delivery and composition in a pervasive environment. The proposed architecture uses external third-party service enablers to automatically compose new services and deliver them to a pervasive environment. Composing new services using available services and deploying them in a pervasive environment makes such environments more attractive to their mostly non-technical users. It also reduces costs associated with offering services in those environments, since human intervention is not required. In this thesis, I review the relevant literature, explain the implementation of the prototype of my architecture and present the results of its performance evaluation. The developed prototype shows the feasibility of my proposed architecture (implementation as well as performance wise) and uses a unique way of deploying a composite service as well as a novel method of applying a workflow repository for service composition.

Acknowledgments

I especially want to thank God who gave me opportunity and the strength to start and finish my PhD. I am also grateful for having Dr. Graham as my PhD supervisor who was the continuous support in different stages of my program. I was delighted to have Dr. Arnason, Dr. Anderson, Dr. Diamond, and Dr. Nikolaidis as my committee members who had to read my thesis several times and helped me to improve my PhD work. I would like to thank all the members of the PDSL lab who gave me invaluable feedbacks during my study.

I could not survive during my long PhD period without financial support that I received from my supervisor, TRLabs, and the University of Manitoba. I am also thankful to Dr. Gole and Dr. Filizadeh (from Electrical and Computer Engineering department) who gave me the opportunity to work with their group and provided me financial support in the most deserving time.

I owe my deepest gratitude to my mother and my father whose supports and prayers are with me since I remember. Undoubtedly, without their support I could not have achieved what I have done so far.

Last but not least my especial thank goes to my beloved wife, Fereshteh, who was with me in all the ups and downs. Without her, it was impossible to tackle the challenges that I faced during my study.

To someone special

Fereshteh

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the past, most devices participating in distributed computing systems were similar in terms of computing power and communication protocols used. Further, the participants normally cooperated to provide a single function. With the introduction of *ubiquitous/ pervasive computing* [85], however, the variety of devices and their capabilities, the heterogeneity of networks connecting them and the variety of tasks concurrently being done have all increased significantly. Moreover, in contrast to earlier distributed environments, the communication links used are now often wireless and the devices are sometimes mobile and may also have limited capabilities. Thus, many heterogeneous distributed computing systems are now comprised of very different computing devices of varying abilities and using different operating environments. Accordingly, the key research challenges are now shifting to focus on providing effective device interoperability.

A wide range of devices including personal electronics (e.g. cell phones, PDAs, etc.), entertainment equipment (e.g. stereos, PVRs, etc.) and even home appliances now have sufficient processing power and abilities to be connected to a network (either directly or using a GD) and therefore may participate in ubiquitous computing systems. Further, multiple such computing devices are commonly found together in pervasive environments.

With the increasing prevalence of wireless mobile devices, the set of available devices and corresponding services that they offer in a given pervasive environment may also change dynamically and sometimes frequently.

The pervasive environment targeted in this research is assumed to be comprised of several network-enabled devices connected together using, potentially, different connection technologies (wired or wireless) and protocols. The underlying functionalities provided by each of the different devices are described as one or more *services*. Services are self-describing and platform independent (with respect to invocation) and can be provided by a physical device directly or via a software program [64]. (For example, a DVD player can offer a "play video" service just as video playing software running on a PC might.) Services can be partially described and characterized by their inputs and outputs.

The ability to automatically discover and combine the services offered by devices in any given environment offers the potential to provide a significantly improved user experience through enhanced (collective) capabilities, increased interaction between device owners, and simplified access to infrastructure and other facilities/capabilities. Unfortunately, this potential is seldom realized because the vast majority of the users of pervasive devices are non-technical and the devices themselves are currently incapable of automatically self-configuring and cooperating to meet users' needs. Since a fundamental tenant of pervasive computing is ease of use, it is becoming increasingly important to be able to allow pervasive devices to discover and inter-operate with one another without direct user involvement.

The number and type of pervasive environments (e.g. homes, meeting/conference rooms, airport lounges, etc.) with a broadband connection to the Internet is increasing rapidly [33]. This suggests the possibility of involving one or more remote third parties in the process of automating the composition of services (i.e. "service composition") to help realize the potential of pervasive computing. Taking this approach not only provides more computational "horsepower" for doing the composition but also provides the opportunity

to measure the effectiveness of known compositions, to exploit their reuse and, when/if necessary, to involve knowledgeable human experts in the composition process.

In this thesis I use a Home Area Network (HAN) [62] as an example of a rich pervasive computing environment, without loss of generality. A HAN is comprised of computing devices, audio/video devices, sensory devices, and networked appliences connected one to another using wireless or wired technology. The connection point between each pervasive environment and the Internet is assumed to be provided by a *Gateway Device* (GD). [1] Normally, the GD is a relatively powerful device (from a computation and, possibly, storage perspective) and can be managed remotely by the Internet Service Provider (ISP) supplying it. This provides a reliable computation source in each pervasive environment and a reliable link from the environment to the Internet. In our assumed environment, there will also be one or more Service Enablers (SEs), that may be distinct from the ISP, providing service management and composition to each pervasive environment. Figure 1.1 shows a simple home area network as an example of a local pervasive environment.

1.1 Motivation and Approach

Due to the variety of devices, communication media, and manufacturers, pervasive environments are highly heterogeneous. Different services (such as video on demand, health-monitoring, etc.) as well as software components used by the participating devices need to be updated regularly to satisfy users' needs and to support interoperability between devices and, possible, Internet-based services (e.g. streaming video with QoS guarantees). With the increasing numbers of such environments and devices in each one, distributing new services and upgrading and composing available ones has become a cumbersome, difficult, and expensive task that requires greater technical ability than most users are capable of.

[1] For a HAN, this device is commonly referred to as a *residential gateway*.

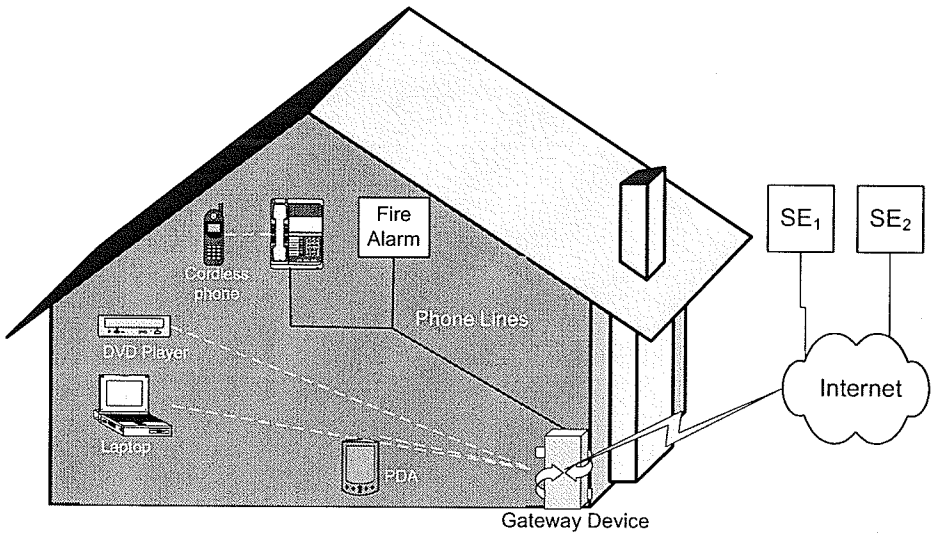


Figure 1.1: A simple home area network as an example of a local pervasive environment

Software development companies try to overcome the existing heterogeneity by developing new protocols or middleware but new standards are slow to evolve and incompatibility between different software components makes the existing service integration and distribution problem more difficult.

In a HAN (or similar) environment, trying to integrate services (provided by devices) in a uniform way requires many issues to be addressed. Among these issues, service discovery and service composition are two related, important, and challenging topics.

Service registration and discovery protocols act as a bridge between devices providing service (devices willing to share their capabilities) and service consumers (client devices which need a service). These protocols therefore not only help devices to advertise and discover services, but also clients to meet their requests. These protocols must also strive to have minimal overhead on the hosting devices.

Composing a new service out of discovered services is a promising solution to help automate the creation of new services for pervasive environments. Unfortunately, when the services available in a pervasive environment are many and their potential interactions numerous, service composition can be expensive and hence impractical for embedded devices to do. This suggests the need for a separate, more powerful entity being involved in composition. SEs can provide not only the required power for service composition but also a shared repository to store commonly used composite services to increase reusability and speed of composition. They can also collect statistics about composite service usage to identify *similar* environments. This classification, as will be presented later in this thesis, can help SEs to send only suitable composite services to a given environment.

The composition of *Web services* has been studied extensively in the literature [65, 77, 54, 68, 42, 27]. Being able to compose a new service using existing ones can provide added value for users. For example, a user might express his/her interest in going to a conference and a composite service comprised of conference registration, flight booking, and hotel reservation could perform the required task, ensuring that all the user's specified constraints are satisfied. Unfortunately, in many pervasive environments, goals cannot be defined, devices offering a variety of services are many, "on-the-fly" (real-time) response is needed, and the user is non-technical and therefore is unlikely to be capable of, or even willing to, direct service compositions. Systems for service composition developed by Sirin et al. [77], Rao et al. [68], etc. involve the user in defining a goal and selecting component services to achieve the goal. Masuoka et al. [56] semi-automatically generate composite services for a pervasive environment but their work is limited in a number of ways. None of the existing work, to the best of my knowledge, is able to automatically find and offer new and useful composite services based only on the available services in a pervasive environment without direct user involvement in somehow pre-specifying the desired service(s).

In this thesis, I propose an architecture to address the challenge of doing composition without direct user involvement. To keep the end-user out of the service composition process, services must be described more accurately. In common with some other work, I use semantic information to describe services' input and output types. Once inputs and outputs have been described using a type ontology, it is possible to discover all the possible sequences of services in which the output of one service can be redirected to the input of another. This approach is called input/output-based matching and is the default mechanism behind the service composition process. Unfortunately, input/output-based matching is computationally expensive. Accordingly, I also use a shared repository of pre-defined abstract compositions (workflows) to speed up the service composition process by "reusing" previously composed services. This helps to address the computational expense of service composition. Other challenges addressed include efficiently reusing previously generated composite services and *ranking* discovered composite services to filter out ones which are less likely to be of end-user interest. Generally speaking, my thesis follows the steps presented in Figure 1.2 to achieve its main goal of supporting effective service composition without direct user involvement.

As it can be seen from the figure, to fully automate the composition process, input/output matching must be employed. This method, however, is computationally expensive and generates many composite services that can overwhelm the end-user. Third party SEs can step in the composition process and provide required computation for input/output matching as well as a storage to keep the generated composite services. The SE also applies a ranking function to generated composite services to identify more useful ones to deploy in a pervasive environments. Later in this thesis different methods of ranking and their impact on the final result of composition will be discussed in detail.

By using an architecture that involves SEs in the creation of each composite workflow and in the deployment of newly created composite services in the pervasive environment(s),

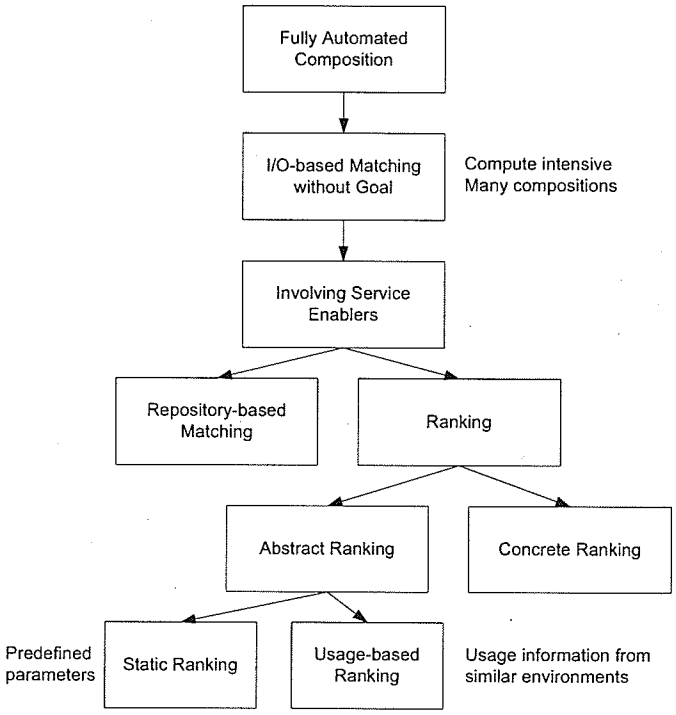


Figure 1.2: Steps in achieving the main goal of fully automated service composition

non-technical users will be able to benefit from composite services without being involved in their creation. Also, ISPs can reduce their maintenance costs using such an architecture by allowing them to do some of their activities (including upgrading software components) remotely.

1.2 Example Scenario

Consider an example scenario that might occur in a typical home network environment. A father has bought a new game console for his seven year old son. He connects it to

the existing home area network and powers it up. At this point, the console is likely immediately capable of performing its primary function (playing games, possibly including Internet-based multi-player games) without assistance from its users. However, the new game console might also be capable of doing tasks other than just playing video games. For example, it might be able to play high definition videos, store and play multimedia files (audio, video, and picture), etc. Expecting either the non-technical father or the young son to be able to exploit the benefits offered by these capabilities is unrealistic.

Using the approach described later in this thesis, the game console will advertise its services in the network and the GD will send information about the newly available services to a SE. The SE, based on the information it has about already deployed devices in the home, will automatically generate new composite services involving one or more of the newly advertised services from the game console. It will then select the most useful composite services and send them back to the home for deployment. Some composite services that could possibly be generated might include:

- Video recording service: The video signal from a TV cable box could be directed to the new game console and the game console could store it as a video file thereby offering a PVR-like service. This new service might initially be run explicitly by a home occupant, or could conceivably be run automatically based on context information indicating that no one is at home when it is time for someone's favorite television show. This sort of video recording service could also be used to store the output of a home monitoring camera as a video file.

- High definition video playback service: The video play back service of the new game console could also be composed with the display service of a high definition TV that already exists in the home.

- Upconverting standard definition video service: The new game console would likely

have sufficient computational ability to be capable of upconverting standard definition videos to better match a home's high deffinition TV. Existing video files on DVDs or even on video cassettes might be sent to the game console for conversion and the recoded output could later be played back on the high definition TV.

In addition to these services, the new game console's services might also be composed with those offered by devices such as digital cameras, video cameras, cell phones, etc. which are commonly available in many homes.

1.3 Contributions

The contributions of my thesis are as follows:

1. A novel service composition model for pervasive environments in general and Home Area Networks (HANs) in particular which involves SEs in the service composition process

2. A novel method for storing workflows (i.e. the SE's shared repository) and an integrated method of semantic matching including both input/output-based and repository-based approaches (described in Chapter 5).

3. Designing a type ontology to help describe services semantically (described in Chapter 4).

4. A method for ranking the automatically generated composite services (both abstract and concrete) to ensure that only the most useful generated services will be offered to the end-user and thereby avoiding "information overload" (described in Chapters 3 and 5.

5. General verification of the ranking scheme (described in Chapter 6)

6. A mechanism for deploying composite services in pervasive environments using OSGi and the UPnP and Jini ubiquitous computing protocols.

7. A novel "OSGi grounding" for OWL-S descriptions and a mechanism for the dynamic creation of such groundings (described later in Chapter 5).

8. A simulation study and scalability analysis to assess the feasibility of supporting many GDs by SEs (described in Chapter 6).

1.4 Results

I developed a prototype system to assess the success of my approach to perform service composition without user involvement. The prototype system, as a whole, was tested with a few real as well as simulated devices. Services offered by these devices were described using the developed realistic type ontology. The system was able to successfully detect devices, identify composite services based on advertised services, and finally deploy them. The composed services are invoked in the same way as other available atomic services are.

I then created many synthetic services, that could potentially be offered by different devices, to test the service composition process more specifically. A program acting as a SE, received these services one at a time and identified all possible compositions using both input/output and repository-based matching. The generated composite services were ranked and at the end (i.e. after all services were received by the SE) statistics about the total number of composed services and number of composed services with a given rank value were collected. By manually inspecting the results of composition, it was clear that the ranking scheme successfully filters out most "undesirable" compositions.

I also performed a simulation to study how my proposed system scales under different

scenarios to assess its practicality in various pervasive environments. The result of simulation shows that the proposed system is capable of handling a very dynamic pervasive environment (such as an airport lounge).

Although the home area network was considered as an example in this thesis, the proposed method for service composition is equally applicable in any pervasive environment such as airport lounges, meeting rooms, etc.

1.5 Thesis Overview

This thesis is organized in seven chapters. After the introduction, Chapter 2 overviews related work in the area of service discovery and composition including service discovery protocols and a number of service composition systems. Chapter 3 describes my high-level service composition architecture for use in pervasive environments that supports a wide range and number of pervasive devices and which can provide composition without direct user involvement. The key components involved in the architecture and their interactions are introduced. To be able to test the proposed system, a domain ontology is required. The design of such an ontology (reflecting real world devices) is presented in Chapter 4. The details of my implementation are described in Chapter 5. Chapter 6 presents the experimental results evaluating my proposed architecture for composition. This chapter also presents a scalability analysis of the proposed system for a variety of pervasive environments. Finally, Chapter 7 concludes the thesis and identifies some areas for possible future work.

Chapter 2

Background and Related Work

Service oriented computing (SOC) [64] is a relatively new paradigm that abstracts different resources as services. This abstraction makes use of resources independent of their platforms. Different capabilities can be used collectively. Using SOC concepts in pervasive computing environments can help developers to hide underlying differences from users.

2.1 Service Oriented Computing

Service Oriented Computing (SOC) is a rapidly growing approach to software structuring particularly for Web-based distributed systems. SOC has evolved to address two shortcomings of existing component-based software development; platform heterogeneity and assumed tight coupling between system components [19]. SOC achieves its goals by using services as the fundamental elements for application development [64]. Although many definitions have been proposed for a service, in this thesis, I assume a service to be a self-describing and platform independent (with respect to invocation) entity that can be provided by a physical device or a software program [64]. For example, a DVD player device can offer a "play video" service just as video playing software might.

Services abstract functionalities and present them to the end-users, as well as to one another, through interfaces defined using standard languages and protocols. Services are not bound to a specific user nor to a particular run-time environment. Software services can also be provided locally or by service providers (running on service enablers), refer again to Figure 1.1. Service consumers (i.e. *clients*), may be either devices or other entities and require no knowledge about the service providers, *a priori.* Services should have the following features [64]:

- Technology neutral: services should be accessible (discoverable and invokable) using widely accepted standards.

- Loosely coupled: services should not need to know about the internal structure of a client nor that of a server.

- Location transparent: services should be accessible without needing to know their locations a priori. This feature can be achieved either by registering service descriptions in a directory or by using broadcast mechanisms in local environments.

A *Web service* is a special and ubiquitous kind of service whose description is available via a URI [64]. Web services are described using XML-based languages and use standard Internet protocols (e.g. HTTP) for communication. Web services are currently the most common instance of the *software-as-a-service* concept [64]. Complex business processes (e.g. loan processing, vacation planning, etc.) and software-based applications can be offered, via subscription or on a rental basis, as services over the Internet. New applications can be constructed by assembling service components (Web services or others) offered by different service providers.

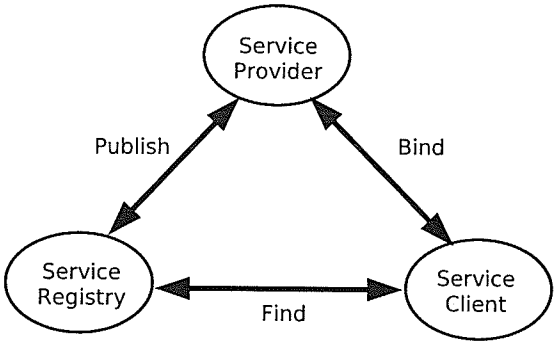


Figure 2.1: The basic service oriented architecture [64]

2.1.1 Service Oriented Architecture

Service Oriented Architecture (SOA) is a way of designing software by organizing available services and identifying relations among them so that different services can communicate and exchange data using standard interfaces and messaging protocols [60] to build software systems from available services. SOA identifies three different participants in a service-based environment: service provider, service client, and service registry (See Figure 2.1). SOA separates the description of a service's interface(s) from their implementation. An interface describes what is provided by a service. The interface should provide enough information to service clients about operations provided by a service to allow clients to invoke them. The implementation of a service can be in any language on any platform.

Using SOA, as compared to traditional software design methods, can provide the following benefits [60]: finding the right services (with respect to a requirement at hand) is faster, replacing a service provider is easier, creating new applications is possible using "service composition", incorporating new services is easier, and supporting new and unpredicted requirements is possible by having an abstract definition of a service.

A Web services platform, for example, provides the necessary facilities (i.e. describing, publishing, and invoking services) to implement an SOA. Some of the key elements of a Web services platform are:

- Web Services Definition Language (WSDL): WSDL [6] describes publicly available functions (interfaces) of a service, their inputs and outputs, binding information about the transport protocol to use, and address information to locate a service. WSDL thus describes a service in terms of messages exchanged in a service interaction [31].

- Universal Description, Discovery and Integration (UDDI): UDDI [8] defines a registry service for (Web) services. Service providers can use UDDI to publish their own services. Service clients then use the UDDI lookup service to discover, and later obtain the description of, their desired services.

- Simple Object Access Protocol (SOAP): SOAP [82] is a message passing and service invocation protocol which normally uses HTTP as its transport protocol. WSDL has built-in support for services accessible via SOAP.

Relying on open-standards elements, a Web services platform allows service providers and consumers to interact in a consistent manner without considering underlying differences such as location and run-time environment. While increasingly common, Web services platform is not the only SOA platform.

2.1.1.1 Ontology-enabled SOA

Ontology-enabled SOA [61] is a combination of the use of semantic information, provided by a domain ontology to describe services, and SOA. An ontology describes different concepts and relations (e.g. subclass, equivalence, etc.) among them. The idea of augmenting

SOA with semantic information has been motivated by the need to infer relationships between two concepts, irrespective of their name or other syntactic information. This enables more accurate service composition.

WSDL describes a service's input(s) and output(s) using XML Schema data types [60] and these data types alone are used to find relations between input(s) and output(s) of two services to determine the flow of a composite service. Also, languages such as WSDL are not capable of describing internal interactions among services. WSDL, just describes what functionalities a service provides and how they are accessible using message passing. To be able to describe the flow of execution within a composite service (i.e. the order of interaction among components of a composite service), we need (among other things) a language to specify the order of messages exchanged between services. However, having such a language only partially addresses the issue since the order of services participating in a composite service must still be specified manually. In dynamic, heterogeneous pervasive environments this requirement could overwhelm end-users and is highly undesirable.

A possible approach to solve both the flow determination and flow description problems is using semantic information to describe services. Machine-readable semantic information about a service makes it possible to discover other services that are "compatible" with a given one. Here, compatibility is defined in terms of the type compatibility of the input(s) and output(s) of two services. Hence, two services are considered compatible if the output of one service can be redirected to the input of another. To be able to find compatible services, their input and output types must be described semantically.

The Web Ontology Language for Services (OWL-S) [30] is a language capable of using types defined in an ontology to describe input and output types of services. OWL-S is also capable of describing the flow between different services in a composite service. By using a common ontology to describe services in a given environment, compatible services can be discovered automatically to create composite services.

Ontology-enabled SOA is particularly appealing in pervasive computing environments where there is no explicit goal to create a composite service. In such environments the users cannot or do not want to, specify what service(s) are desired. They simply want such services to be provided automatically. Having semantic information about services, composite services can be automatically created and then made available to users without their direct involvement.

2.1.2 Introduction to Pervasive/Ubiquitous Computing

Ubiquitous computing, also referred to as pervasive computing, is a vision of computing introduced by Mark Weiser [85]. In his vision, computing devices should "weave themselves into the fabric of everyday life until they are indistinguishable from it". Current advances in hardware, software, and networking technologies make Weiser's ideas feasible and practical. In this section I review the historical roots of pervasive computing. Then, I explain the pervasive computing model and major "actors" in a pervasive computing environment. Finally, I link the use of SOA to developing pervasive applications.

2.1.2.1 Evolution of the Pervasive Computing

Pervasive computing can be considered an evolutionary product of the integration of two related areas, namely distributed systems and mobile computing [74]. As a result, a pervasive computing environment is both heterogeneous and dynamic in nature. Distributed computing arose with the popularity of personal computers and the development of networking technology. Once many computers became connected, the idea of sharing resources became important. Distributed computing provided users access to shared resources (e.g. storage, printers, files, computing, etc.) without requiring users to have explicit knowledge about the physical location of resources.

The challenges in building a distributed system, which are also relevant to pervasive computing, are to provide remote communication, fault tolerance, high availability, and security [74]. Although there are solutions to these challenges for distributed systems, they are not, in many cases, directly applicable to all pervasive computing environments where there may be no pre-configured network infrastructure and possibly no powerful computing devices capable of acting as different "servers" (e.g. name server, file server, etc.).

Mobile computing is another area related to and influencing pervasive computing. Mobile computing was born, mainly, with the introduction of mobile devices (e.g. cell phones, laptops, etc.) and wireless communication (e.g. cellular and local area networks). Mobile computing supports the idea of *anywhere, anytime* computing. Some challenges, related to mobile computing, include mobile networking (ad-hoc protocols), energy awareness, and location sensitivity [74]. These challenges are also relevant to pervasive computing where many of the devices may be tiny, resource limited, and sometimes mobile (due to the mobility of their operating environments such as cars, human bodies, etc.).

2.1.2.2 Pervasive Computing Model

In a pervasive computing environment four different components (actors) commonly exist [72]:

1. Devices: different devices with different computation and communication capabilities can participate in a pervasive environment. The devices can be considered to be the core of the pervasive computing.

2. Ad-hoc networking: an ad-hoc network connects potentially many devices "on-the-fly" without predefined infrastructure. The available networking protocols and models for traditional networks (e.g. the client/server model) must be tuned or even re-designed to be applicable in such pervasive environments.

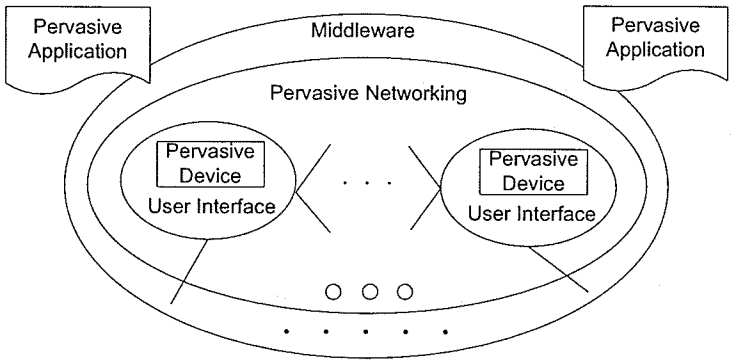


Figure 2.2: Pervasive computing model: pervasive devices can talk either directly, if they know each other's protocol, or via middleware (adapted from [72])

3. Pervasive middleware: Some middleware layer typically needs to be available in a pervasive computing environment to present the devices (i.e. network core) functionalities to the end-user applications. Middleware can also abstract underlying functionalities by advertising them as "services" allowing application developers to use Service Oriented Architecture (SOA) to structure their software. This provides a homogeneous view of an underlying network of different devices.

4. Pervasive applications: a pervasive application is the contact point between a pervasive computing environment and end-users. The application should operate according to the end-user's expectations and must require minimum user involvement to *hide* computing devices, protocols, etc. as much as possible in the environment.

The relation between different components of a pervasive environment is shown in Figure 2.2 where the middleware layer acts as an umbrella over the potentially heterogeneous network of devices.

2.1.2.3 Using SOA for Pervasive Computing

SOA is a useful approach to designing pervasive applications where heterogeneity and dynamism are important characteristics. The loose coupling feature of services can help to solve the heterogeneity problem by allowing standard interfaces irrespective of underlying device or execution platform. A device, in a home for example, can publish its provided services using standard interfaces. Another device can then discover and later use the published services without knowledge of the implementation details of the invoked service.

Another typical characteristic of pervasive environments is their dynamism in terms of addition and removal of devices and their corresponding services. In such environments, applications cannot be deployed *a priori*. Different environments, according to their current occupants and available devices, may require very different applications to run. SOA is a promising potential solution to allow new applications to be dynamically created by assembling existing services.

A networked home, as an example of a pervasive environment, has the potential to benefit from SOC. Currently, homes may be equipped with different sensors, computing, and entertainment devices and they will have more in the future [15]. Further, homes typically no longer use only one vendor to provide all the devices they need. Devices, manufactured by different vendors, may use different protocols which is an impediment to inter-operation. However, many useful applications (e.g. printing a picture from a digital camera, etc.) require interaction between multiple devices which makes developing such applications both important and challenging. By using services, as the fundamental components of interaction in such environments, we can use service composition to dynamically create such new applications.

To be able to exploit the benefits of SOC in a pervasive environment, there must be some middleware providing an interface between the devices and service technologies (i.e.

description and messaging). Pervasive devices will, normally, communicate using their predefined (sometimes proprietary) protocols and thus, middleware must exist which understands the various protocols and which can be used to publish device functionalities through a standard that is understood by other service clients and providers. Such middleware can also be used as the foundation for deploying new (composite) services as described later.

2.1.3 Service Oriented Computing and Failure Handling

Services in SOA can become unavailable due to reasons such as device failure, network failure, etc. In a pervasive environment, service failure can happen more frequently as a result of participating mobile devices. Failure, in this context, refers to the unavailability of a part of a composed service (e.g. sub-service, device, or network) due to mobility or incorrect behaviour of a service component, for example due to overloading. To handle a failure, it must first be detected. Then, the detected failure can either be tolerated or reported. In this section I briefly review different failure types and modes. Then, I briefly discuss failure detection and tolerance techniques applicable in pervasive environments.

Shankar et al. [75] classify the types of failures in a pervasive (non SOA-enabled) environment into the following categories:

- Device failure: a device can fail due to power loss, broken part, etc.
- Application failure: applications, running on devices can fail as a result of bugs in their code, operating system errors, unhandled exceptions, etc.
- Network failure: participating devices in pervasive environments are often wireless and can be temporarily disconnected from the network due to low signal strength, heavy communication traffic, or low link bandwidth. Network failures may sometimes be incorrectly detected as device failures.

- Service failure: there are a number of essential supporting services in a pervasive environment such as registry and discovery services. A service can fail for a number of reasons including bugs in the code, operating system errors, and errors in sensing or inferring events.

In an SOA-based pervasive environment, there is no difference between an application and a service since devices offer their functionalities as services. A service, in addition to the reasons mentioned for application failure, can be unavailable if a SE stops offering that service for any of a number of reasons such as upgrading to a new version or changing access permissions. Therefore, service failures will be a superset of application failures.

Gorbenco et al. [39] identify different *modes* of failure of Web services. Their identified modes, which are also applicable in pervasive environments, are:

- Transient failure: this mode of failure can be tolerated/handled using generic techniques such as rollback and retry.

- Non-transient failure: this is a permanent failure that can be detected and later handled by redundancy.

- Evident failure: this type of failure is easy to detect using classical techniques such as timeout.

- Non-evident failure: this type of failure requires a special detection mechanism, typically supported at the application level. In this type of failure, the faulty component commonly generates erroneous data.

Table 2.1 shows the relation between types and modes of a failure.

Table 2.1: Relation between types and modes of failure

Component Type	Failure Mode			
	Transient	Non-transient	Evident	Non-evident
Device	√	√	√	
Service	√	√	√	√
Network	√	√	√	

2.1.3.1 Failure Detection

Detecting a failure in a pervasive environment is not necessarily an easy task. Mobile and resource-limited devices as well as unreliable communication links make it difficult to correctly detect a failed service in a timely fashion. Timeout, a classical approach, is considered to be an easy and, in most cases, effective way of detecting some failures. To be able to use the timeout method, *heartbeat* messages are normally used to keep track of alive devices [75]. Each device periodically sends an "I am alive" message to a fault detector. However, this method does not scale well, since the messages can create considerable network traffic when many devices exist in an area. Another limitation of the timeout technique is its inability to detect service failures unless each service, hosted on a device, sends heartbeat messages separately which makes network traffic higher.

Another approach to failure detection is associating a *lease* time with each service. If a service (or host device) fails to renew its lease before expiration, the service is considered to be unavailable. Service clients, on the other hand, can use lease time to specify the amount of time they need a service. If they fail to renew their lease, the requested service can be considered free. Although renewing leases also requires exchanging extra messages, the number of messages can be minimized if the lease time is assigned carefully [20].

Heartbeat messages normally detect a fault, shortly (depending on the timeout period) after its occurrence. On the other hand, lease mechanisms postpone the detection of a failed

service until the point of invocation or lease renewal. It is, of course, possible to trigger an event when a lease expires but this will put more overhead on the service registry service.

In addition to failure, a device can become unavailable due to its mobility. A mobile device can join and leave a network easily. The unavailability of a mobile device should be detected and subsequently handled. The handling of failure due to mobility may have to be treated differently from other failures.

2.1.3.2 Fault Tolerance

Once a failure (device, service, or network) is detected, depending on the type of failure, it may be possible to tolerate it. The simplest approach to tolerating a service/application failure is *restarting* the failed service/application. Restarting is possible if the device hosting the service/application is still operational. If a service fails as a result of device failure, a similar service running on another device needs to be found. This type of fault tolerance is achieved through replication either explicit or by chance (e.g. a DVD player can co-exist with a computer having a DVD drive to provide redundancy). Both service and device replication are possible. For example, video playing software on a desktop computer and a DVD player can both offer a "play video" service. Also we can have two printers offering a "print" service. In either case of replication, an appropriate replica must be discovered when a failure is detected. To provide a smooth transition to a new service, the state of the failed service can be stored on some stable storage which can then be used by the replica service to *resume* the failed service's work at or near the point of failure.

If a failure cannot be tolerated (i.e no replacement or prefered service/device is available), it must be reported to the user properly [75]. Reporting a failure depends on the severity of failure and, possibly, the location of a user. For example, if a user is in a room with an appropriate display device, a good choice for reporting an error to the user might be showing an error pop-up on the display device. Whereas, if no one is in a home and a

fault happens, sending an e-mail to the home owner or calling him/her (depending on the urgency of the fault) might be a better choice. This is a substantial problem which is not in the scope of this thesis.

2.2 Service Discovery

Once we have services in an environment, we need to be able to discover and use them. Thus, service discovery becomes an important issue in SOC. Also different services can be aggregated, using service composition techniques, to provide new functionality. In this chapter, I will, first, describe service discovery principles and provide a representative sample of available service discovery protocols. Then, I will review service composition techniques and systems developed mainly for pervasive environments.

2.2.1 Service Discovery Principles

Service discovery enables clients, looking for services, to find their desired services. Also it allows devices, providing a service, to advertise their capabilities [80]. In general, the following actions take place in the context of service discovery:

- Service Announcement,
- Service Lookup, and
- Service Invocation

2.2.1.1 Service Announcement

Each device must be able to announce its capabilities if it wants to share its resources. Further, a service should be presented in an appropriate way so that subsequent lookups

can find it easily.

Service representation can be done in a programming language dependent or in a language independent way. Programming language dependent methods (e.g Java based) use a specific language's capabilities to specify a service. On the other hand, programming language neutral methods (e.g. XML based techniques) specify a service in a format which is not restricted to any specific language(s).

Common approaches are to either use a directory, to register available services' descriptions, or to use a periodic broadcast mechanism to announce services. For small networks with a limited number of devices, broadcast or multi-cast methods can work well. However, for larger networks with many devices such mechanisms may impose unacceptable network traffic load.

A service directory can be centralized or distributed. In the distributed case, one or more directories exist in the network and reside on different machines and the directories may be organized in either a hierarchical or a flat fashion. This method provides some level of fault tolerance by removing the single point of failure present when using a centralized directory. However, each client has to use multicast or broadcast to figure out the addresses of directories. In the centralized case, only one directory exists in the network and all the nodes can easily and quickly find the location of this directory (either by pre-configuring the directory's address or by broadcasting a message). The service directory in this approach might, however, become a bottleneck and, in the case of failure, the rest of the system will not be operational.

2.2.1.2 Service Lookup

Regardless of the service announcement technique used, a client must be able to find its desired services. A lookup service tries to find a match between available and requested services. Different factors such as proximity to a resource, context information, or QoS

issues may be considered in selecting a match [87]. Some lookup services try to find an exact match based on the client's request and the service's attributes [87]. (Each service may have several attributes and these attributes should use a standard naming convention.) Some others return a list of possible matches to the user so he/she can select the best service. Some proposed lookup services also try to compose a client's requested service from multiple existing ones [66]. To increase the accuracy of a service discovery protocol (i.e. the probability of finding a service), semantic information about the inputs and outputs of a service can be used in the discovery phase [26]. (Additional details about the semantic description of services is given in Section 2.3.1.1.)

2.2.1.3 Service Invocation

Once a client finds its desired service, it must be able to use it. Thus, the client needs a mechanism to invoke the discovered service. Some service discovery protocols (e.g. the Bluetooth Service Discovery Protocol) do not have invocation mechanisms and the client has to use another protocol after locating a service. Most service discovery systems, however, do provide an invocation mechanism as well. The result of discovery might be a simple address which is subsequently used to access the resource. In some cases an interface object might be returned to the client instead while, in other cases, a complete object is downloaded to the client.

2.2.2 Service Discovery Systems

Different service discovery protocols have been implemented to handle the heterogeneity present in dynamic networked environments (e.g. pervasive environments). These protocols try to connect existing devices and appliances seamlessly regardless of what systems and, in some cases, lower level protocols are being used to provide them. Furthermore,

they must be concerned about scalability to be able to handle the addition and removal of numerous devices from a network. In this section I will briefly review the most common service discovery middleware used in home area and related networks. This illustrates the diversity of the protocols available which complicates the practical aspects of supporting automated service composition.

2.2.2.1 Jini

Jini [84, 10], introduced by Sun Microsystems, is network middleware that provides APIs for system developers that facilitate the creation of dynamic distributed applications. Jini uses a registry to register available services in the environment and this registry is used for discovery purposes.

Jini runs on top of a Java Virtual Machine (JVM) and consequently provides a machine independent environment. (Jini can run wherever there is a JVM available.) Each device in a Jini-enabled network is represented as an object and each device can provide services to other devices. Jini uses Java's Remote Method Invocation (RMI) capability for inter-object communication.

All of Jini's services are distributed and each device must have the capability to communicate with the registry to be able to participate in a Jini network. There is also a *surrogate architecture* and a version of Jini, *Jini Mobile Edition (JiniME)*, which enable non-Jini devices (i.e. devices with limited resources which are unable to run a full JVM) to be connected to a Jini network [43]. These are explained in more detail in the next section.

A lookup service is needed to provide the required information about available services in the network. Each server (the device which wants to provide a service) sends an object representing its service to the lookup service using Java's object serialization mechanism. A client that needs a specific service sends its request to the lookup service which, if there is a suitable registered service, sends the appropriate *service proxy* object to the client [84].

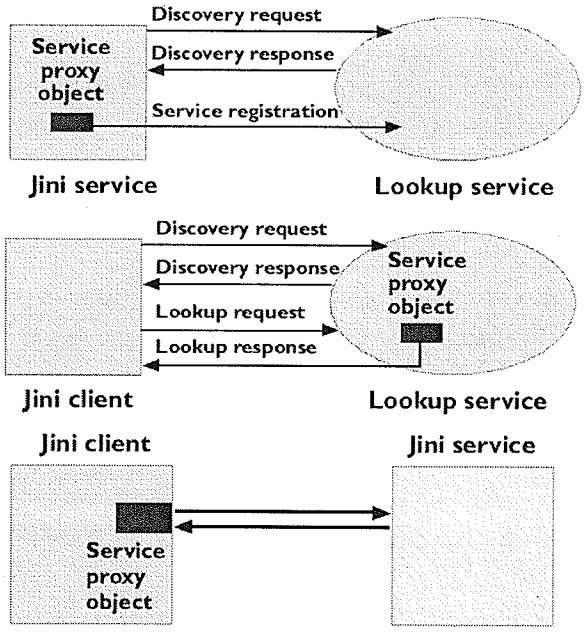


Figure 2.3: Lookup and discovery in Jini (adapted from [84])

The sequence of registering and using a service in Jini is depicted in Figure 2.3.

Due to the distributed nature of Jini, a lookup service can be run on any machine. Thus a basic discovery service introduces the lookup service to a newly added device. Different lookup services can also exist in the network to provide a level of fault tolerance [43]. A server will register its own services with as many lookup services as it discovers. Thus, a client accessing any of these lookup services can find a reference to the desired service.

To handle network unreliability, Jini uses the *lease* concept [84, 43] as follows. Each resource is leased to a client and the client has to renew the lease to hold it for another period of time. Also services, at the time of registration, are given a lease time. If a service wants to be used, it should renew its lease before expiration. When a service becomes unavailable/inaccessible (e.g. due to network failure) all authorized users' leases expire.

This mechanism guarantees that a client can access any service that it discovers as long as the service is accessible.

The Jini Surrogate Architecture

The Jini surrogate architecture [43] enables non-Jini devices to participate in Jini networks. In this architecture, a Jini-capable surrogate host is connected to the non-Jini device (normally a small embedded device with limited resources) to provide Jini services to this device. The surrogate host, naturally, must be powerful enough to run a JVM.

Each device has a surrogate Java proxy object (similar to a service proxy). When a non-Jini device joins a Jini network it registers its surrogate object with a lookup service using the surrogate host. Communication between a non-Jini device and a lookup service takes place through the surrogate host. The communication between such a device and its surrogate is done via a predefined protocol and is transparent to other Jini devices communicating with the non-Jini device.

JiniME

Another solution to connecting resource-limited devices to a Jini network is using Jini Mobile Edition (JiniME) [49, 43], which was developed at the Rochester Institute of Technology. This solution, compared to the surrogate architecture, does not need an established Jini network with a surrogate host. While Jini uses Java 2 Standard Edition (J2SE) for its runtime environment, JiniME uses Java 2 Micro Edition (J2ME) [43] for this purpose which is able to run on many resource constrained devices. In J2SE, object movement between different JVMs is done using object serialization and the marshaling and unmarshaling processes are done automatically using Java libraries. These libraries need a considerable amount of memory and processing power. In contrast, J2ME lacks this feature so JiniME

uses separate libraries and forces the programmer to do the object marshaling manually. This results in a far smaller memory footprint for this version of Jini.

Each JiniME device hosts a lookup service and a class-file server and registers its own services with the local lookup service [43]. Each lookup service has a service proxy object which is downloadable by other devices to access the lookup service and subsequently the services offered by a given device.

A *Jini bridge* can also be used to connect a JiniME network to a Jini network [43]. This device must be able to talk with both sides and hence it must run J2SE as well as J2ME. A Jini bridge detects new devices in the JiniME (usually, wireless) part of the network and creates corresponding objects to advertise the provided services to the Jini network part. A similar service is provided to the wireless part for services in the wired network by the Jini bridge.

2.2.2.2 UPnP

Universal Plug and Play (UPnP) [59] is a "service discovery protocol" developed by Microsoft which uses existing Internet protocols (such as IP and TCP) to connect different devices. Unlike Jini, UPnP does not use a registry to keep track of existing services. There are three types of entities in a UPnP network [81]:

1. Services,
2. Devices, and
3. Control points.

Services are the smallest controllable entities in UPnP [9]. Devices can be physical or logical. Physical devices are hardware components that may contain multiple logical devices or services. Logical UPnP devices can be implemented in any programming language and

on any operating system [9]. They use an XML-based device schema to inter-operate with each other. The XML-based description file contains information about the type of input parameters that a device accepts, and output values that a device returns after doing its task. Devices which can not implement TCP/IP can participate in a UPnP network using a proxy. Control points control and discover other devices in a UPnP network.

UPnP defines six steps in a typical interconnection as follows [59]:

1. Addressing: Each device must have an IP address. It gets its IP address from a DHCP server. In the case of failure to get an address, a client is assigned an address in the default 169.254/16 range [9]. After choosing a random address from the aforementioned range, the client checks for a conflict and if the address is already in use, it tries another one until the conflict is resolved. Once a temporary address is obtained, the client continues to check periodically to find a DHCP server.

2. Discovery: After being added to the network, a device introduces itself to a control point. (A control point is a device which provides interaction with a client who needs to use a service.) Devices use a discovery protocol based on the Simple Service Discovery Protocol (SSDP) [38] and multicast discovery messages (including a device type and a URL pointing to the corresponding device description). Interested control points, listening to standard multicast ports, get notified about new services. Similarly, when a control point is added to a network it multicasts an SSDP discovery message (including a description of the desired service) to search for a new service. Devices which have a matching service (according to the disseminated description) will reply to this message.

3. Description: A control point uses the URL provided in the discovery message described to find more information about the device. Described in XML format, a

UPnP description is divided into two parts: device description and service description. The device description contains vendor specific information (e.g. model name and number, etc.) and the service description contains the service name, a URL to the service description, a URL for control and a URL for event handling. (These URLs provide addresses to which a command for execution or event notification request should be sent. These are described further in the "Control" and "Event handling" bullets described next.) A service description also includes a list of actions the service responds to, and arguments for each action.

4. Control: After getting the information about a device and its provided services, a control point can send an action to a service to be executed. Control messages are expressed in the Simple Object Access Protocol (SOAP) format. SOAP uses XML and HTTP for implementing remote procedure calls and UPnP uses SOAP to send control messages to control URLs (provided by the description message) and return results and errors back to control points [9]. SOAP is comparable to Java's RMI technique for executing remote methods. However, SOAP is language independent and is based on XML.

5. Event handling: A control point can also subscribe itself to receive a notification when the state of a service changes. The control point sends a subscription message using General Event Notification Architecture (GENA) [29] messages to a specified device's service. If the subscription is accepted, the device sends a unique identifier as well as an initial event message containing all event variables (i.e. variables whose values will be changed upon receiving an event) back to the control point. After this step, when an "evented" variable changes, subscribing control points are notified by GENA NOTIFY messages sent over HTTP [59]. When a control point needs no more event notification, it may cancel the subscription.

6. Presentation: Some devices may also have a presentation URL in addition to a description URL. Having this URL, the control point can retrieve a page from this address and, depending on the capabilities provided by the page, a control point can allow a client to control the device and/or view the device's status via the corresponding HTML page.

2.2.2.3 HAVi

Home Audio Video Interoperability (HAVi) [7] is a special purpose networking standard used to connect different audio and video devices to each other. HAVi provides an environment in which different audio and video devices from different manufacturers can communicate.

HAVi is language and operating system neutral. In a HAVi network there is no master node. Instead, each node can control the others. Nodes can also be placed anywhere in a HAVi network. HAVi uses the high bandwidth digital IEEE 1394 (a.k.a. "FireWire") network to physically connect the audio and video devices.

HAVi is comprised of a set of software components implementing such basic functionalities as device abstraction, network management, event management, resource management, and inter-device communication. These components provide APIs to allow users to build a distributed application on a HAVi network.

Each device in HAVi is modeled as an object-based software component [7]. Objects have well-defined interfaces through which the services they provide are accessible. Each host provides an execution environment in which a service can run. Services can be provided either by device manufacturers or by third parties. Each object has a system-wide unique identifier assigned by the HAVi messaging system.

A HAVi system also maintains a registry in which each object registers itself. The registry offers an interface for clients to query for specific services. HAVi uses message

passing for inter-object communication. The implementation details of message passing can differ from device to device and between vendors as well. However, the message format is determined by the HAVi specification. Before sending messages, objects must find each other using the registry. One object can then send a message to another.

Each device in HAVi is represented and controlled by a software component called a Device Control Module (DCM). DCMs provide flexibility for devices joining and leaving the system. When a device joins or leaves a system, a DCM is installed or removed from the network, respectively, and an event is fired. The events in a HAVi network are handled by an event manager component which is responsible for event delivery. Moreover, for each *controllable* function inside a DCM, there is a Functional Component Module (FCM). FCMs are similar to logical devices or services in UPnP. Depending on the device, a DCM may contain different FCMs. For example, an entertainment unit might have tuner and display components which are both represented as FCMs.

2.2.2.4 OSGi

The Open Services Gateway initiative (OSGi) [17, 55] provides a Java based framework in which different services, offered by different devices, can be integrated. For portability and scalability reasons, implementation of a service is separated from its interface(s). Thus, depending on the environment in which a service is being deployed, different implementations may be chosen.

A *bundle* [55] is a package (deployed as a Java ARchive (JAR) file) for service implementation. All bundles have a similar structure specified using the OSGi standard. (e.g. they must all have *start* and *stop* methods.) A bundle contains the required resources for implementing zero (e.g. a utility bundle that only provides some packages for other bundles) or more services and a *manifest* file containing headers that specify the required parameters for installing that bundle. The OSGi framework provides a mapping from services to their

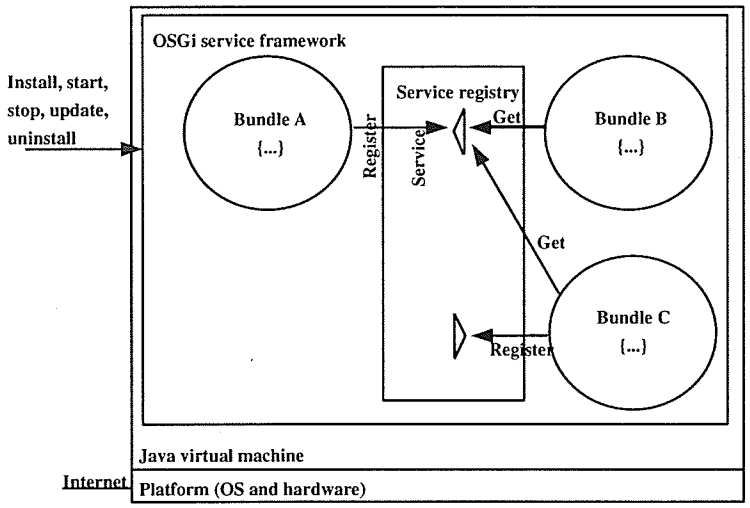


Figure 2.4: Bundle service registration and subscription (adapted from [55])

implementations.

One of the installed bundles, the *initial bundle*, is started first and it then manages the life-cycle (installing, starting, stopping, and uninstalling) of all other bundles [62]. Moreover, each bundle has a special class, *bundle activator*, instantiated by the framework, with stop and start methods for stopping and starting bundles, respectively.

Once a bundle is installed, it registers its services which can then be discovered and used by users as well as by other services to accomplish their tasks (see Figure 2.4). The OSGi framework is responsible for dynamically installing and updating bundles. This feature of the framework allows installed bundles to be extensible even after installation.

OSGi can generate and handle three types of events [62]:

1. ServiceEvent: This event is generated upon registration, de-registration, or changing of a property of a service.

2. BundleEvent: This event is used to report a change in the lifecycle of a bundle

3. FrameworkEvent: This event reports the start of the framework. It also reports any errors at framework startup time.

To handle a generated event there is a corresponding listener interface for each type of event. Bundle implementers can use these interfaces to catch and handle events of interest.

One of the interesting goals of OSGi was to provide interoperability among different middleware (e.g. Jini and UPnP). Such interoperability can be achieved by installing appropriate *driver* bundles in the OSGi framework. A driver bundle (e.g. Jini driver) imports all the services advertised outside of the OSGi framework, using the corresponding middleware (in this case, Jini), as OSGi services. Once the services are imported into the OSGi framework, they can be used by other services within OSGi. In this way, different services advertised by different protocols can make use of each other within the OSGi framework.

A driver bundle can also export native OSGi services (provided by different bundles within OSGi) using its supported protocol (e.g. Jini) to the outside of the OSGi framework.

2.2.2.5 Obje

Obje [11] is a service interoperability framework developed at the Xerox Palo Alto Research Center (PARC). Obje tries to provide interoperability between different devices and services by allowing them to "teach" each other new media formats and transport protocols. Devices and services use mobile code (e.g. a Java program) to deploy a new protocol or media format on another device.

The Obje framework defines a few general agreements, called *meta-interfaces* that each device involved in an Obje network (a.k.a. an Obje-enabled device) must implement [11]. For example, consider a meta-interface called *Data Transfer* that an Obje-enabled music jukebox and an Obje-enabled music player have implemented. When these two devices

want to communicate with each other, the Data Transfer meta-interface establishes a connection and defines a media format between the two parties based on the network and device's capabilities respectively. Once this connection has been established, the jukebox can teach the target music player the network protocol and media format by sending mobile code to the music player. The player can then play the music after "learning" the protocol.

2.2.2.6 AutoHan

AutoHan [73] is a complete home area network solution (mainly based on UPnP) connecting AutoHan compliant devices together. AutoHan uses a UDP version of the HTTP protocol for data transfer and XML for describing services and other entities. The three main components of AutoHan are: a directory service named DHan that is used for service discovery, an event handling system based on UPnP GENA, and a module named IHan for accessing AutoHan from the Internet.

DHan, AutoHan's directory service, manages the available devices and their services. Each device acquires DHan service using broadcast messages and subsequently registers its properties and functionalities using DHan. DHan provides a lookup service so that different devices can use each other's services. Similar to Jini, AutoHan uses a lease mechanism to handle unavailable objects and services. Each registered service is assigned a lease that automatically expires unless the service renews its lease.

Control in AutoHan is based on events. AutoHan uses, and extends UPnP GENA to send and receive events. If a device does not support the GENA event system, AutoHan provides a proxy to convert a non-compatible event to GENA.

To access devices in AutoHan from the Internet, a software module named IHan is used which accepts user requests from the Internet via HTTP, authenticates them, and if they are valid, forwards them to other parts of AutoHan. Before forwarding requests, IHan extracts GENA and DHan headers from the relevant HTTP request. IHan does the reverse of this

operation when a reply is sent to the user. This feature enables a user to control in-home devices remotely.

2.2.2.7 @HA

The At Home Anywhere (@HA) [81] project is a home appliance integration project from the University of Twente, in the Netherlands. In this project, home appliances, based on their capabilities and resources, are divided into three categories (from low to high capability):

1. 3C appliances: very simple devices which only have basic network connection capability

2. 3D appliances: simple devices implementing a network protocol stack as well as a service discovery protocol

3. 300D appliances: powerful devices with a memory size bigger than 1MB and an embedded computer. The most powerful 300D appliance, available in a HAN, is selected to provide a service directory. This is done using an election algorithm.

The service discovery protocol in the @HA project has two important features: resource awareness and robustness [81]. To address resource awareness, the protocol selects a service directory, called *Central*, from the 300D devices. However, if such a device does not exist, the protocol will instead work in peer-to-peer mode without a central node. When a 300D device starts, it broadcasts a message to announce its presence. If there is another 300D device in the network, it also broadcasts the same message and the central node election algorithm is initiated. Otherwise, the single, new 300D device becomes the central node. Once a central node is elected, it broadcasts a message and asks all other nodes

to register their available services. If a device becomes available after a central node is selected, it broadcasts a message to find the central node.

Robustness, in the @HA service discovery protocol, is provided by selecting a backup for the main service directory. In the directory selection algorithm, the second most powerful 300D device is selected as a backup directory. The main directory and its backup communicate at specific intervals to detect failure of each other. In the case of the failure of either device, device handover takes place and the failed device is replaced by the other. Depending on the type of failed device (main directory or backup) a different handover algorithm is run.

2.2.2.8 The Service Location Protocol

The Service Location Protocol (SLP) [45] is a language independent service discovery protocol developed by an Internet Engineering Task Force (IETF) working group. There are three different software entities (a.k.a. agents) in this protocol: User Agents (UAs), Service Agents (SAs), and Directory Agents (DAs). The protocol can operate both with and without DAs. A DA's address can be configured either statically (i.e. by entering it manually, reading it from a file, or hard-coding it) or dynamically (i.e. using DHCP) [44]. If these methods do not work, SAs and UAs can also try to find a DA (if any exists) using multicast messages [44]. If a DA exists, SAs register themselves with the DA and UAs can then unicast their queries to the DA to find suitable services. An SLP network may have several DAs but there is no synchronization mechanism between them [44]. When there is no DA in the network, UAs use multicast to find services. Any SA which can provide the desired service, unicasts a response to the UA.

SLP only provides a service lookup mechanism (location and contact information) [80]. A client has to use some other method (which is determined by the implementer) to actually invoke the discovered service(s).

2.2.2.9 Bluetooth SDP

The Bluetooth protocol stack [50] has a built-in mechanism for service discovery called the "Service Discovery Protocol" (SDP) [67]. This protocol was designed to operate in a dynamic environment such as ad-hoc networks. It is capable of matching services by type and attributes and browsing available services without a priori knowledge about them [67]. Each device, providing a service, has to run an *SDP server*. For each service provided by the device, a service record is kept on the SDP server. A client sends an SDP request to the server to find a service. The server searches the service records and, in the case of a match, it sends a *service handle* to the client [81]. Bluetooth SDP does not provide a mechanism to send commands to the target service or to invoke remote methods. Thus, the discovered services must still be accessed through other protocols built on top of Bluetooth's basic communications facilities.

2.2.2.10 SSDS

Secure Service Discovery Service (SSDS) [32] was an early attempt to integrate a discovery protocol with security. It was developed as part of the Ninja project at the University of California, Berkeley. Although SSDS is implemented in Java, it uses XML, rather than Java objects, to describe and locate services. Its target was enterprise-wide distributed applications trying to find and use resources in a secure way. It uses both symmetric and asymmetric encryption [79] (also known as secret and public key encryption, respectively) for data confidentiality. It also uses message authentication code (MAC) [79] to guarantee message integrity. A MAC is an authentication tag (a.k.a a checksum) obtained by applying the secret key to a message. To provide authenticated and encrypted communication, SSDS uses a custom re-implementation of Java RMI.

There are three main components in the SSDS architecture [32]: clients, services, and

Secure Discovery Service (SDS) servers. SDS servers use authenticated multicast messages to periodically announce their presence to the system. Described in XML, a service, after finding an appropriate SDS server, sends its description using an authenticated and encrypted one-way broadcast message. A client connects to an SDS server using authenticated RMI and sends the requested service's description, presented in XML, to it. Subsequently, the SDS server searches the available services, stored in an XML database called XSet [40], to find a match. In the case of failure, the SDS server might, based on its configuration, send the request to other SDS servers [32].

To support the dynamic addition and deletion of SDS servers, the available SDS servers can be organized as a hierarchy. When the load of a server exceeds a specified threshold, it spawns a new child and delegates a portion of the load to the new server. New servers, like their parents, will announce their presence using multicast messages. Parent servers keep track of their children by using "heartbeat" messages.

2.2.3 Summary of Service Discovery Systems

Table 2.2 summarizes the key features of some main service discovery protocols. As discussed earlier, some of these protocols (e.g. Jini and UPnP) have mechanisms for both discovery and invocation of a service. While, others (e.g. SLP and Bluetooth SDP) only enable clients to locate a service. Clients must then use another mechanism to access the located service.

2.3 Composition of Services

Service composition is a technique used to create new services using smaller, existing ones. The composition of services is usually addressed in two stages:

1. Composite service creation: in this stage a user's composite service request is converted to a "composite service template" (a.k.a. a service workflow). This stage is similar to parsing a database query or a computer program. At this stage possible service components and their control flows are identified. Normally, a user specifies his/her desired input(s) and output(s) and a service composition system tries to find a sequence of services accepting the user's input(s) and producing the user's output(s). Different methods such as modeling services as graph nodes, using semantic information about services, etc. can be used to find "Input/Output" (I/O) compatible services.

2. Composite service execution: in this stage, a composite service template is mapped onto an execution path. Service composition frameworks execute the template and access the physical services and at the same time try to execute the composite service efficiently (from different points of view such as bandwidth consumption, response time, etc.). This stage requires "service discovery, integration, and execution" [27].

Generally, we can define the following steps in a service composition process [69]:

1. Service Description: All providers of services must describe their provided services in such a way that searching for them becomes as easy as possible. Different XML-based languages such as Web Service Definition Language (WSDL) and DARPA Agent Markup Language for Services(DAML-S[1]) [2] have been used for this purpose.

[1]A newer version of DAML-S is the Ontology Web Language for Services (OWL-S) which is described later in Section 2.3.1.1

2. Description Translation: Sometimes two different languages are used for the external (user-side) and internal (system-side) representation of services. In such a service composition framework, some component must take the responsibility for translating between the two representations.

3. Workflow Generation: A workflow generator tries to find one or more valid sequences of available services fulfilling a user's request. The generated workflow(s) will subsequently be used for execution.

4. Workflow Selection: If the number of generated workflows (i.e. valid workflows satisfying a user's request) is more than one, an evaluation is carried out based on certain user preferences (e.g. non-functional attributes such as cost and/or performance) and the top ranked workflow is selected for execution.

5. Workflow Execution: The selected workflow is executed in this step. The workflow can be executed by the framework which generated it or the workflow may be fed to another system for execution.

2.3.1 Composite Service Creation Systems

In this section I review some of the attempts to create a system for automatic (or semi-automatic) service composition. The reviewed systems either accept a user's composite service description (in terms of input(s) and output(s)) and try to find a workflow based on the available services or help the user to build his/her own desired workflow. The fundamental assumption in these systems is the existence of a (semantically described) goal and the need to match between the requested and advertised services. What is really needed in pervasive environments, however, is a system that does not require any goal defined by the end-user and which finds all the possible composite services based on the available ones.

To remove the unnecessary compositions (those that are unlikely to be of end-user interest) from the list of generated composite services, it must also be possible to accurately rank the compositions and then filter out the low-ranked compositions.

Semantic service description and service matching are reviewed in some detail first.

2.3.1.1 Semantic Service Description

Services need to be described to enable subsequent discovery. The more information you can provide about a service, the more effective service discovery can be. Most of the service discovery protocols presented in Section 2.2.2 use syntactic information (e.g. name, simple data type, etc.) to describe a service's input(s) and output(s). One method to increase the probability and accuracy of discovery is to use semantic rather than just syntactic information to describe a service. To describe a service semantically, we need an *ontology*, more specifically a domain ontology, to describe the different concepts which are available in a specific domain and which are useful for describing services. Some details of designing such an ontology for a pervasive environment are discussed in Chapter 4.

The Web Ontology Language (OWL[2]) [12] is the W3C recommendation for specifying an ontology. Based on XML, OWL is a language that allows ontology developers to describe new concepts (i.e. anything that information must be collected about) and relationships between them. Once the desired concepts are described, a language is required to describe services operating on those concepts. The Web Ontology Language for Services (OWL-S) [30] is such a language and it exploits OWL's capabilities and provides markup language constructs to describe a service based on its input(s), output(s), precondition(s), and effect(s).

The OWL-S description of a service is composed of the following parts:

[2]OWL is, in fact, an acronym for Web Ontology Language

- service profile: explains what a service does in terms of its inputs, outputs, preconditions, and effects.

- service grounding: explains how to access the service. A service grounding describes where the implementation of a described service can be found and what protocol(s) should be used to invoke the service's implementation.

- service process: explains how a service works. It shows the sequence of operations, iterations, decisions, etc. A service process effectively glues together a service profile and a service grounding. The service process also indicates whether a service is composite or atomic.

OWL-S, by describing a service using these three different aspects (profile, process, and grounding), provides a separation between service description and implementation. This separation means that other parties (such as device manufacturers as well as service creators) can provide semantic descriptions of services. Also, it means that a service description can have multiple implementations corresponding to where the service is going to be deployed.

2.3.1.2 Semantic Matching

Once we define a service semantically using OWL-S constructs, we can use semantic matching [63] to find *similar* services as well as those that are exactly the same. Semantic matching relaxes the normal, "exact" matching concept by considering hierarchical relations which may exist between the input and output types of two services.

For example, assume that we have an advertised service A and a requested service R. For an exact match, we will say service A matches Service R if the inputs of the two services as well as their outputs have exactly the same types, respectively. On the other hand, using

semantic (or "partial") matching, service A matches R if there is a relation (based on an ontology defined in a language such as OWL) between the inputs and, respectively, the outputs of the two services. Using this concept, Paolucci, et al. [63] introduce four different levels of matching: *exact, plugin, subsume, and fail.*

Paolucci et al. match the outputs of the two services first. Let us assume that O_A is the output type of an advertised service and O_R is the output type of a requested service. If $O_A = O_R$ then O_A and O_R are the same concept and the match will be *exact.* If O_A subsumes O_R (i.e. O_A is a superclass of O_R in an ontology hierarchy) then the advertised service provides more than what is requested and O_A can be *plugged-in* in place of O_R and the match type will be *plugin.* If O_A is a *direct* supercalss of O_R, the match will be assumed to be an exact match and this relation is called "direct subsumption". If the super/subclass relation is not direct, it is called "indirect subsumption". On the other hand, if O_R subsumes O_A then the requester needs more than the advertised service provides and the match type will be *subsume* indicating that only part of the requested functionality is provided by O_A. If there is no subsumption relation between O_A and O_R, the match will fail. This process is then also applied to the inputs of the two services and the final result of matching is determined based on the matching results of the outputs and inputs. If both the input and output types of two services are the same, the two services will match. The level of matching between two services is determined by the level of matching between the input and output types.

2.3.1.3 Semantic-based Service Composition Systems

In this section, I review semantic-based service composition systems. As the reviewed systems do not all have specific names I refer to them based on who their developers were.

Sirin et al. [78] proposed a semi-automatic solution for service composition. Their implemented system has two components: a composer engine and an inference engine

with a KnowledgeBase (KB). Written in OWL-S, service descriptions are converted to RDF (Resource Description Format) [13] triples [3] and loaded into the KB. The composer system then helps a user to create a workflow step-by-step.

A composition starts when a user selects a registered service. A query is then sent to the KB to retrieve the selected service's inputs and for each input another query is generated to retrieve (using the inference engine) all the possible services providing this input. The results of queries are shown to the user to choose from. Since the result of matching might produce many services, a user can also define filtering rules to make handling the results of queries more manageable. Matching can be done exactly or "generically". If an output of a service, say S_1, is the same as an input of another service, say S_2, those services are exactly matched. If there is a subsumption relation between the output of S_1 and S_2's input, the two services are said to be generically matched.

The result of composition is realized as an OWL-S document (using the composer engine) specifying the order of the involved services. The new service can also be advertised and composed with other services. After this document is generated, execution of the workflow can be done by invoking each service and passing data according to the user selected workflow.

Rao et al. [68] proposed a framework to automatically generate a DAML-S [4] workflow from a user's request. The user's request (i.e. inputs and expected output(s)) is also described in the DAML-S language. The proposed system uses a Linear Logic (LL) [68] theorem prover to find a logical (i.e. input/output compatible) sequence of available services fulfilling the user's request. First, the description of all available services as well as the user's request are translated from DAML-S into LL. The theorem prover accepts descriptions in LL and tries to find a valid sequence of services capable of doing a requested

[3]Each RDF triple is composed of a subject, an object, and a predicate which is normally represented as a graph whose nodes are the subject and the object. The predicate, links the nodes.

[4]DAML-S is the older version of OWL-S.

task. The output of the theorem prover is a workflow presented in process calculus [68]. Then, a translator converts the generated workflow into a DAML-S or BPEL4WS (Business Process Execution Language for Web Services) [1] document. To the best of my knowledge, this system can not execute the generated workflow but it can be fed to any system accepting DAML-S or BPEL4WS.

Another system, proposed by Majithia et al. [54], translates a DAML-S composite service description into a BPEL4WS document. Their system stores two types of workflows, *abstract* and *concrete*, with the goal of increasing re-usability. An abstract workflow does not have any information about service instances and its service specification is high level. A concrete workflow, on the other hand, is a mapping from an abstract workflow onto available services. Majithia et al.'s system is similar to Rao's but they do not convert DAML-S descriptions to Linear Logic and, instead, use an inference engine for DAML-S to carry out subsumption reasoning.

SWORD [65] is a toolkit for service composition. Unlike other service composition systems, SWORD models a service based on its inputs and outputs using Entity-Relation (ER) modeling. Users specify initial and final states (i.e., inputs and outputs) of a composite service and SWORD finds a sequence of rules, using its rule-based plan generator, satisfying the request.

Each service has two types of inputs: 1) conditional inputs (a.k.a preconditions) specifying conditions on input types and the relationship to other services, and 2) data inputs on which a service operates [65]. Similarly, each service has conditional outputs (a.k.a postconditions) and data outputs. For a composite service, the preconditions will be the conjunction of all conditional inputs of the involved services. The same thing holds for the postconditions of a composite service.

2.3.1.4 Graph-based Service Composition

The relation between inputs and outputs of services can be modeled as a directed graph. In such a graph, nodes represent services and edges show the compatibility between the output and input of two services. Using such a notation, the problem of creating a composite service can be modeled as finding a path in a graph. The composition methods reviewed in this section use this idea to create composite services.

Arpinar et al. [18] assume that services are described in DAML-S. They create a weighted graph showing the relations between different services. Nodes in this graph represent services and two nodes are connected if the output of one service can be fed into the input of the other (considering both exact and partial matching). The weight on a link is the result of applying a function on quality and similarity measures between the input and output types of the two services. Arpinar et al. also extend the concept of a simple graph to include services with multiple inputs and outputs.

A request specifies the input and output types for a desired service and the Bellman-Ford algorithm [18] is used to find the shortest path between the request's input and output types, if one exists. A user can give different weights to similarity and quality rates, depending on his/her requirements, and based on these values, different paths can be selected from the graph. Arpinar et al. do not describe how to add a new node to the graph and how to update the appropriate links when a new node is added.

Kalasapur et al. [47] use a graph, called a "service graph" to represent services. Their associated service composition technique is called SeSCo [48]. Nodes in the service graph represent (sub)components of a service required for execution of a service. Edges between nodes represent the order of (sub)components. Nodes, as well as edges, can have their properties (e.g. name, location, etc.) assigned by different functions. Once the description

of individual services are ready (i.e. their service graphs), another graph called a "community graph" [5] is created. The community graph has a different structure than a service graph in which nodes represent parameter (i.e. input or output) types. For each service graph, the input and output types are added (if such a type has not been added before) as new nodes to the community graph. An edge is also added to the community graph to link the newly added node(s). Kalasapur et al. also assume that a request for a service is represented as a graph called a "request graph". Similar to the service graph, nodes in the request graph are (sub)components of services and edges show the order of (sub)components.

Given the community graph, an algorithm tries to find a set of services matching the request graph. First, the algorithm tries to find a match (from the community graph) for each node i of the request graph r, V_r^i. If such a service does not exist, the algorithm locates nodes corresponding to the input and output types of node V_r^i in the community graph (V_{comm}^{in} and V_{comm}^{out}). Successfully locating these nodes and a path between them in the community graph shows an existing composite service matching V_r^i. If a service or a composition is found for each edge in the request graph, it means that matching was successful.

Hashemian et al. [46] use a directed graph, similar to Kalasapur et al.'s community graph, whose nodes are parameter types and edges show services converting an input type to an output type. Interface Automata (IA) [46] are used to represent inputs, outputs, and the dependencies (i.e. $i \rightarrow o$) between them. The dependency information is represented using a collection of dependency sets in which each set specifies relations between an input and an output. Dependency sets provide a mechanism to know how two operations should be executed. For example, if dependencies corresponding to two services appear in the same dependency set, those two services must be performed sequentially.

A request is also represented using IA in terms of its inputs, outputs, and dependencies.

[5] This graph is called "property graph" in [48].

To discover the possible services capable of fulfilling a request, Hashemian et al. apply Breadth First Search (BFS) on all nodes appearing on the left hand side of at least one dependency. For a dependency like $i \rightarrow o$, all nodes reachable from i, represented as a path, will be returned using the BFS algorithm. The algorithm will stop if it finds o, otherwise it will run the BFS until it terminates without returning o. In the next step, dependencies and corresponding returned paths will be examined to find possible compositions. If the length of a path corresponding to a dependency is one, it means that we have an atomic service capable of generating the requested output and we do not need a composition. Otherwise, the services appearing in the path define a composite service. Finally, the result of a request is represented as a binary expression tree.

2.3.1.5 Summary of Composite Service Creation Systems

A summary of the characteristics of a representative sample of the reviewed systems that try to create a composite service given a user's request is shown in Table 2.3. The "semantic matching" feature, in the table, indicates whether or not the reviewed system uses semantic matching to find a composite service. The language to describe a service is identified in the "description language" feature. The user involvement in the service composition process is indicated by "low", if the user only enters the requested service's input(s) and output(s), and "high", if the user has to choose components of a composite service. The ability of a system to execute the created composite service is determined by the "service execution" feature.

2.3.2 Composite Service Execution Platforms

In this section I review some representative service composition platforms that, given a pre-existing composite service workflow described based on its components, try to execute

this workflow efficiently, possibly with some non-functional constraints (e.g. QoS, etc.). Some of the reviewed systems in this section are no longer available or supported but are included for completeness.

2.3.2.1 ICARIS

The Infrastructure for Composability At Runtime of Internet Services (ICARIS) [83] project tries to support service composition in a distributed environment. This work is based on the authors' previous work on dynamic software component upgrading at run-time called *software hot-swapping* [34]. Services, their inputs, and outputs are described in XML and service discovery is done using Jini's service discovery mechanism.

ICARIS provides three different approaches to composing services [83, 24]. In the first method, *interface fusion*, a composite service *interface* is created by extracting the signatures of available services and combining them together as a single interface. Later, at the time of using the composite service, individual services incorporated in the composite service are called in the predefined sequence. In this case we may (or may not) have input/output dependencies between involved services.

In the second, standalone, approach unlike in the interface fusion approach, an actual composite service is created (not just an interface description of it). In this method components are independent and they are connected together using a method called "pipe and filter". In this approach ICARIS creates some "connection services" [83] which retrieve the output(s) from one service and send them to the input(s) of another service.

In the third, stand-alone with a single body method, the code and logic (overseeing the sequence of execution) of the component services are inserted into the composite service. In other words, composable methods from different component services, involved in the composition, are extracted and assembled into a single body. This method eliminates the message passing and invocation cost. However, the method is more complicated and they

require more time to complete than using the previous methods. This method requires access to the source code of methods and, naturally, generated composite services will get bigger as the number of services involved in the compositions increases.

2.3.2.2 eFlow

eFlow [21, 22] is a service composition system developed at HP laboratories. eFlow integrates heterogeneous services (e.g. services on the Web) to make a composite service. The specification of a composite service is given by the user and eFlow models the composite service as a process represented by a graph [24]. This graph has service, decision, and event nodes [21]. Service nodes represent a basic or composite service invocation. Decision nodes control the order of execution and event nodes enable services to signal and respond to various events (e.g. notification of the completion of a service, suspension of a service, etc.). Also, an eFlow graph supports "transactional regions" [24] which identify portions of the graph that must be executed in an atomic fashion.

The main components of eFlow are:

- Composition engine: The composition engine is the main component of eFlow which handles composite service requests. It processes all completion messages received from service nodes and subsequently schedules the execution of the next service node(s) in the graph. It uses a service discovery mechanism to find the actual services that correspond to service nodes.

- Service discovery service (broker): A service broker finds the actual service components that can fulfill service node requests.

- Basic services: Component services that are the building blocks for service composition.

Service graph nodes have associated "service selection rules" specifying the desired service's characteristics. Service selection rules are specified in a language that is recognized by the service broker. Usually an XML-based language is used to describe service selection rules and services as well. Based on the service selection rules, the service broker finds an appropriate service satisfying the needed criteria. The service broker, as a result, returns to the composition engine a document in XML specifying the discovered service's location, its input and output information, etc. eFlow users may define their own service broker and plug it into the system. In this case the language for specifying service selection rules must be compatible with the new service broker.

eFlow provides repositories of processes, service nodes, and data type definitions to increase re-usability of the system [22]. A process in the repository of processes can be complete or just a template which allows plugging in of service definitions by a service designer. Service node and data type repositories provide storage for definitions of services and data types, respectively, that are frequently used to compose services in a given environment. Definitions of service nodes and data types are structured into groups and group hierarchies for easy access by service designers. The repositories in eFlow, however, are not used for automatic composition of services. Furthermore, no mechanism for updating the repository is presented by the developers.

To be more flexible, eFlow also supports a concept called "dynamic service flow modification". Dynamic service flow modification allows eFlow to dynamically modify the sequence of service components in a composite service. This feature may be required to handle unexpected exceptions, to incorporate new business policies, etc. Modifications can be done in either *ad-hoc* or *bulk* mode. An ad-hoc change is applied only to a specific composite service whereas, in contrast, bulk changes affect all existing composite services. Dynamic service flow modification is controlled by authorization rules. Only authorized users are allowed to make such modifications.

2.3.2.3 The Ninja Service Composition Platform

The Ninja [40] project at the University of California, Berkeley, developed another service composition platform for use in wide area networks hosting a wide range of clients on different devices (from cell phones to desktop computers). The Ninja system uses service composition to enable different clients with varying resources and network accessibility to use the same network services. For example, different clients can request the "play video" service, provided by a host in the network and the Ninja system will deliver the service in different formats, by composing it with other services to do operations such as transcoding, based on the capabilities of the client device, the existing bandwidth between the client and the original SE, etc. Ninja uses SSDS (discussed in Section 2.2.2.10) as its service discovery protocol.

The main concept in Ninja is the *path*. A path chains different *operators* together using *connectors*. Operators are services doing computation on a passing data flow. Described in XML, each operator has attributes and a type. The operator's description helps the system to find a logical combination of operators that can do a user requested task. Operators can be either *long-lived* or *short-lived* [40]. Long-lived operators are registered and located using a service discovery mechanism. Short-lived operators (a.k.a. dynamic operators), on the other hand, are created dynamically by the system to do necessary transformations and are transparent to the users [40]. A service capable of converting a high resolution video stream to a low resolution one might be implemented as a short-lived service. Connectors abstract away the details of underlying links and connect operators residing on different nodes.

The main functional component of the Ninja system is the Automatic Path Creation (APC) service. The APC service accepts a user's request, identifies required components,

discovers the services, sends information to the component services, and monitors the execution flow.

To create a path, a user provides the APC with a specification of the endpoint nodes in the path, a partially-ordered list of operators that must be included in the path, and some cost-related criteria that the path should satisfy [40]. Path construction is a four-step iterative process as follows [40]:

1. Logical path creation: The APC searches the XML descriptions of operators to find logical sequences of operators capable of fulfilling the user's request. The result is a list of sequences sorted in descending order based on the user's optimization parameters. As this list may be large (if the number of operators is high), only a subset from this list is selected for subsequent processing. After running the physical path creation phase (the next step), it might be necessary to choose another subset of the already created logical paths if the selected paths do not satisfy the user's requirements.

2. Physical path creation: A physical path is a mapping from a logical path onto physical nodes that are capable of running the required operators. Nodes for long-lived operators are located using service discovery. Nodes for short-lived operators are chosen based on the nodes' capabilities (CPU, memory, etc.). If none of the constructed physical paths meet the user's requirements, a different set of logical paths are selected (step 1) and this process is repeated.

3. Path instantiation and maintenance: The APC service selects the best possible physical path (based on the user's inputs) and sets up the required dynamic operators (i.e. short-lived operators) and appropriate connections between nodes hosting operators. Once the connections have been established, data flow is started. After that, a *control*

path between the APC service and the operator nodes is set up. This path is used for gathering performance information as well as for error reporting.

4. Path tear-down: Once a path is no longer needed, the user asks the APC service to remove it. The APC service stops the data flow, removes connectors, and frees resources. It can also cache path information for subsequent re-use.

Due to the importance of path creation, Ninja runs the APC service on a cluster of workstations to provide fault tolerance and scalability through load distribution.

2.3.2.4 Task Computing

Task Computing [56, 57] tries to fill the gap between tasks (i.e. what a user wants) and services (what is offered by devices). Task Computing tries to enable non-expert users to accomplish their tasks (which may be composed of several services) using services which are available in their environment (e.g. in a meeting room, etc.).

Task Computing assumes the availability of semantically described services on the Web (i.e. Web services) and/or offered by different devices. The high-level and semantically rich description of services allows Task Computing to do the required service composition to fulfill a user's request. Once the required services for a particular task are found, Task Computing uses UPnP to interact with actual devices providing services.

A Task Computing Environment (TCE) includes the following components [57]:

- one or more Task Computing Clients (TCCs)
- one or more semantically described services
- one or more semantic service discovery mechanisms
- optionally, one or more service controls (which enable a user to create, temporarily hold, and remove services)

One Task Computing client, called the Semantic Task Execution EditoR (STEER), provides a user interface showing what is available in a user's environment. STEER uses the UPnP discovery protocol to find underlying services (e.g. "display on a projector", "local file storage", etc.). Once a service is found, STEER either retrieves its semantic description (if there is one) or creates a description based on pre-defined templates. The semantic description is then used for subsequent composition. STEER has an inference engine as well as an execution engine that are used to create and subsequently run composite services.

When a user wants to compose a service, he/she runs STEER from his/her browser. STEER shows the user all possible compositions (limited to only two services and starting with no-input services) that can be done in the user's environment. In this step, the user can decide either to execute one of the proposed compositions or he/she can manually build more complex compositions starting with one of the suggested compositions. Finally, when a user wants to execute a composite service, STEER invokes each component service and redirects the output of each service to the input of the next.

For example, assume that a user wants to execute a composite service including "display on projector" and "local file storage" services. First, STEER locates the user's preferred local file (e.g. by showing a list of files to the user to choose from). Once the file is retrieved, STEER sends this file to the projector which is presented as a UPnP device. The projector device receives the file from STEER and displays it.

STEER plays a broker's role in composition and monitors the execution process. To the best of my knowledge, however, Task Computing does not provide any storage of newly created composite services for future use. A user's computing devices (e.g. laptop, PDA, etc.) are also required to run STEER which places a burden on resource-limited devices such as PDAs.

2.3.2.5 InterPlay

InterPlay [58] tries to integrate devices in a networked home using two concepts: *pseudo sentences* and *task sessions*. A pseudo sentence, in the simplest form, has a verb, a subject (content type) and target device(s) to describe a user task (e.g. "play" (verb) "the movie M" (content) "on living room TV" (target device)). Different parts of a pseudo sentence are selected from pre-defined lists. Once the user task is defined using a pseudo sentence, InterPlay maps the task to a sequence of invocations of services running on UPnP-enabled devices.

A task session captures the current execution state of a task like "playing", "paused", etc. A session is a dynamic concept and devices can be added/removed to/from a session. For example a second TV can be added to a DVD playback session to show a DVD on two TVs. InterPlay uses OWL (Web Ontology Language) and Resource Description Framework (RDF) [4] to describe devices as well as user tasks. The Semantic description of tasks helps InterPlay to find appropriate devices on to which to map the task at hand.

2.3.2.6 SpiderNet

SpiderNet [42] is a peer-to-peer service composition framework developed at the University of Illinois at Urbana-Champaign. Its main emphasis is on guaranteeing decentralized QoS-aware service composition in a service overlay network [42]. SpiderNet is implemented as distributed middleware that maps a user's composite service requests onto available distributed services. Service composition requests consist of a "function graph", showing service functions and their relations, as well as certain QoS requirements. The function graph in SpiderNet is a directed acyclic graph and SpiderNet can change, if required, the order of nodes in the function graph to meet the user defined QoS constraints.

SpiderNet maps a function graph onto a "service graph" whose nodes are services,

found by a service discovery process and the links are mapped onto the available overlay network links. SpiderNet uses the Pastry Distributed Hash Table (DHT) [71] for service discovery. Each service that wants to register itself, generates a hash key based on its function name by applying a secure hash function. A device or client, on the other hand, who wants to find a service, generates a hash key in the same way and sends a query for the service using this key to the "target service " which the DHT routes to.

SpiderNet uses a novel method called Bounded Composition Probing (BCP) to collect resource and QoS information about its contributing providers. A node that wants to provide a composite service (e.g. streaming video service which may consist of finding a server, enhancing the image, and playing components) sends a limited number of *probe packets* to the destination node in the service graph. Probe packets are processed in each passing node and contain the user's QoS requirements. The destination collects all probe packets and, based on the gathered information and the user's QoS requirements, selects one path and sends an acknowledgment message back along the reserved path (i.e. the service composition graph) to the sender.

SpiderNet assumes that each service component runs continuously and accepts data for processing and outputs some data to send on the network. Thus, service invocation in SpiderNet is done by sending data to the first service in the composition chain and from that point on, other services will be invoked automatically. SpiderNet also maintains a small number of backup service composition graphs and in the case of node failure in one graph, it switches to an alternate graph.

2.3.2.7 Broker-based Service Composition

Chakraborty et al. [27] at the University of Maryland, Baltimore County, proposed a distributed broker-based protocol for service composition in pervasive/ad-hoc environments.

In their protocol, clients select a *nearby* broker dynamically and delegate the service composition task (i.e. discovery, integration, and execution) to the selected broker. This mechanism evenly distributes the service composition task among potential brokers, achieves load balancing, and avoids single point of failure problems. It is assumed that composite services are pre-described in DAML-S and that the order of subcomponents is known.

The broker selection phase is followed by the *service discovery* phase. Service discovery is done using Group-based Service Discovery (GSD) [25]. GSD is a peer-to-peer service discovery protocol capable of doing semantic service matching. Services are described in the DAML-S language and GSD does not need a directory to register available services. Each node in a GSD-based network, keeps a cache of *nearby* advertised services and forwards its cache information to others based on the existing group hierarchy in the DAML-S specification of services. This selective forwarding property of GSD, generates a controlled number of broadcast messages and results in better use of network bandwidth.

After finding appropriate services, the *service integration* phase is started. Service integration deals with combining discovered services and filtering out unnecessary components based on execution-level cost estimates. In this phase an Execution-level Service Flow (ESF) [27] is generated. An ESF contains the addresses of actual services (an ESF can be considered to be a concrete workflow) and also network parameters (such as bandwidth, number of hops, etc.) and finally specifies the execution flow of services.

Once an ESF is created, the actual execution takes place during the *service execution* phase. In this phase, the broker coordinates the execution of services in the order specified by the ESF. The broker sends the information received from a previously executed service to the next service in the ESF. Since many component services and computing nodes can be involved in a composite service, it is possible to experience failure of services in the workflow and consequently failure of the composite service execution process. To take advantage of partial results from services executed so-far, Chakraborty et al. use a

checkpoint-based fault tolerance mechanism during the execution phase [27].

2.3.2.8 CoSMoS

Fujii et al. [36] propose a system to dynamically compose services. Their system allows a user to describe a composite service as well as its semantics in an intuitive form (e.g. a natural language). As an example, a user might enter "print direction from home to restaurant" as a request to get directions from his home to a restaurant. The semantics of the requested composite service (i.e. its operation, its inputs and outputs, etc.) are inferred and converted into a *semantic graph* to be used as a template in the composition process.

Fujii et al. use the Component Service Model with Semantics (CoSMoS) to capture the semantics of component services. CoSMoS defines a component by specifying its operations and properties. CoSMoS models the semantics of a component by using *concepts*. Concepts represent abstract ideas (e.g. "direction") and actions (e.g. "print"). The semantics of a component's operations, inputs, outputs, and properties are annotated by concepts. CoSMoS uses its semantic graph representation to describe operations, properties, and concepts of a component service in a machine-understandable format. Once the semantics of the requested composite service as well as all the component services are described using the semantic graph representation, a service composition mechanism, called Semantic Graph-based Service Composition (SeGSeC), discovers appropriate component services based on the semantics of the requested composite service.

SeGSeC has four different modules: 1) RequestAnalyzer, 2) ServiceComposer, 3) SemanticAnalyzer, and 4) ServicePerformer. The RequestAnalyzer parses the natural language description of the user request and converts it into a CoSMoS semantic graph representation. The output of this module is consumed by the ServiceComposer which discovers component services based on the user request. The result of this module is a workflow including the discovered component services. The ServiceComposer starts generating the

workflow by finding a component service that performs the same operation as specified in the user's request ("print" in the above example). Once the first component is discovered, ServiceComposer discovers other services that provide inputs to the initial component service and adds them to the workflow. This process continues recursively for all the newly added services that require an input. An input and an output are considered compatible if their data types as well as their associated concepts are compatible.

The generated workflow is sent to the SemanticAnalyzer which examines the semantics of the generated workflow to make sure that it satisfies the user's request. For example, in the example of getting directions from home to a restaurant, the SemanticAnalyser will discard a composition if home is selected as the destination. If the generated workflow is approved by the SemanticAnalyzer, the ServiceComposer passes the workflow to the ServicePerformer. The ServicePerformer executes the generated composite service by invoking the operations of the component services.

Fujii et al.'s results show that their approach to service composition is not scalable. Although they evaluated their implemented prototype with very few component services and only simple compositions, the number of nodes in the semantic graph is relatively high which can potentially be a bottleneck as the number of component services increases.

2.3.2.9 Synthy

Synthy [14] is a Web service composition platform that requires a user request for composition. Synthy is a multi-stage composition platform that performs service composition at two levels: 1) logical composition and 2) physical composition. The logical composition involves only Web service *types* to create a logical workflow (a.k.a. a template). Web service types are abstract and do not correspond to any implementation. The physical composition, on the other hand, includes Web service *instances* to generate an executable workflow (a.k.a. workflow instance).

Synthy has three different modules: 1) the *Logical Manager*, 2) the *Physical Manager*, and 3) the *Runtime Manager*. When a request arrives, the Logical Manager generates K templates that meet the functional requirements of the request. To select K templates from all possible templates, Synthy uses *Hamming distance* among the templates to select the templates in such a way that their service types are maximally different from each other. (This decreases the number of failed workflows due to the failure of one service). These templates are passed to the Physical Manager to create executable workflows by selecting appropriate instances for each component service. The Physical Manager uses the non-functional requirements of the request to select service instances. Synthy uses *cost, response time,* and *availability* of a service instance as selection criteria. The generated executable workflows are passed to the Runtime Manager which selects the best workflow (in terms of overall composite QoS) to execute.

2.3.2.10 Summary of Composite Service Execution Platforms

The features of the reviewed platforms for service composition are summarized in Table 2.4.

2.3.3 Shortcomings of Existing Systems

Tables 2.3 and 2.4 show that most of the reviewed service composition systems and platforms have targeted either Web services or infrastructure-based networks and assume the existence of a target service and powerful computing nodes. While this work is of interest, the methods are less directly applicable to pervasive environments for the following reasons:

- In pervasive environments, unlike Web services, "on-the-fly" composition must be done by non-technical users so it is impractical to assume that a specific target service can be defined by the user. Involving a user in the composition process is highly unattractive in pervasive computing where a primary goal is hiding the details of using computing devices from the end user.

- The breadth of different devices possible in pervasive environments normally means different vendors and most likely different protocols for communication will be the norm. This makes creating composite services a more challenging task and one where the value of being able to re-use previous compositions is high.

- The required computing resources for matching (in particular semantic matching) are often beyond the capabilities of resource limited devices that are commonly found in pervasive environments. This suggests the need to involve a third party which has enough computing power and other resources to support the matching process.

To address these issues, I propose a service composition approach which is more suitable for pervasive environments. The proposed approach tries to automatically find possible new composite services driven by the discovery of newly advertised services. With this approach, a user does not have to specify a desired composite service but can, instead,

be offered a list of possible composite services ordered by expected usefulness to choose from. This minimizes end-user involvement in the composition process.

To move the overhead of service composition off of resource limited devices, my approach to composition involves third party (SEs) that have adequate resources (computationally as well as technically) to do service composition. Such SE can store predefined as well as generated abstract composite services in a repository and share this information among different pervasive environments to permit rapid and automated re-use. In addition to such re-use, the SE can, when necessary, generate composite services on-the-fly using input/output matching. To remove undesirable generated composite services, the SE employs an intelligent ranking scheme to remove low-ranked composite services from the list of generated ones. SEs can also use the shared repository to collect statistics about composite service use in similar environments to identify "recurring" usage patterns. Such information can then be exploited to offer only those composite services that are likely to be of end-user interest in a given pervasive environment. Although eFlow stores processes and services, it only helps service designers to design new composite services and are not used for automating the service composition process.

To deal with the heterogeneity of a pervasive environment, I have built on an interoperability framework designed to handle different communication protocols. I also avoid introducing any new protocol or changing any existing ones by advertising newly created composite services in terms of all the available protocols.

2.4 Composite Service Execution

Execution of a composite service (a.k.a. service orchestration) is the process of invoking components of the composite service based on an order defined in a workflow. The execution is performed by a "workflow execution engine" (or an "orchestration engine"). In

the execution phase, instances of component services must be accessed and invoked. Invocation of a service may require providing correct number of parameters. The result of invoking a service is normally sent as an input to the next service in the workflow.

There are two main patterns in executing a composite service workflow: star and mesh [27]. In the star pattern, data and control flow between component services resemble a star. In other words, there is a central entity which is responsible for invoking component services with required parameters. It also receives output of an invoked service. Mesh-based approach, on the other hand, does not involve a central entity to shape data and control flow. Each service provider is responsible for invoking the next service in the workflow.

No matter which execution pattern is selected, there are some issues related to execution of a composite service that must be considered:

- Parallel execution of services: the workflow of a composite service is not necessarily linear and parallel execution of component services is possible. The workflow execution engine must provide required mechanisms (e.g. synchronization, etc.) to accurately run service instances in parallel.

- Concurrent accesses to a service instance: services offered by devices such as TV in a home are network, can participate in many composite service workflows. If more than one of such workflows run at the same time, it is possible to have concurrent accesses to the same instance of a service. The workflow execution engine must handle these conditions to correctly execute different workflows.

- Atomic execution: different components of a workflow may have atomicity requirements. For example, booking a flight and reserving a hotel room as two separate components of a "vacation planer" service must be executed as a transaction. In other words, if one of these component services fails, the whole composite service

must fail. The workflow execution engine can provide such mechanisms to run transactional workflows.

- Failure handling: services involved in composite service workflows can fail or become unavailable. The workflow execution engine must provide required mechanisms for detecting and later handling failed services. This feature is more important in pervasive environments where most of devices are mobile.

Table 2.2: Comparison of different service discovery protocols

Feature	Jini	UPnP	HAVi	SLP	Bluetooth SDP	SSDS	@HA
Announcement /Lookup	Directory	Multicast	Directory	Directory /Multicast	Broadcast	Directory	Directory /Broadcast
Service Matching	Interface-based	Syntax	Syntax	Syntax	Syntax	Syntax	Syntax
Service Invocation	Mobile code	SOAP	Vendor Dependent	N/A	N/A	N/A	XML parsing
Architecture	Client/Server (C/S)	P2P	C/S	C/S or P2P	P2P	C/S	C/S or P2P
Security	JVM-based	No	Access Control /Signature	No	No	Auth. /Encryption	No
Directory Address	Configured /Multicast	N/A	Configured	Configured /Multicast	N/A	Multicast	Broadcast
Number of Directories	One or more	N/A	One	One or more	N/A	Multiple	One with backup
Leasing Concept	Yes	Yes	Yes	Yes	No	No	Yes
Programming Language	Java	Independent	Independent	Independent	Independent	Independent	Independent
Applicable to pervasive env.	Yes	Yes	Yes	Yes	Yes	No	Yes

Table 2.3: Comparing composite service creation systems

Features	Sirin et al.	Rao et al.	SeSCo	My System
Semantic matching	Yes	Yes	No	Yes
Description language	OWL-S	DAML-S	Graph-based	OWL-S
(semi)Automatic	Semi	Auto	Auto	Auto
User involvement	High	Low	Low	No
Service execution	Yes	No	No	Yes
Target environment	Web	Web	Pervasive	Pervasive

Table 2.4: Comparison of different service composition platforms

Feature/ System	ICARIS [83]	eFlow [21]	Ninja [40]	SpiderNet [42]	Broker-based [27]	Task Com-puting [56]	InterPlay [58]	CoSMoS [36]	Synthy [14]	My Sys-tem
Area of Applica-bility	Pervasive	Web ser-vices	WAN-based	WAN-based	Ad-hoc	Pervasive	Pervasive	Web ser-vices	Web ser-vices	Pervasive
Needs user's request	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No
Deploying compos-ite service	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Multi protocol	No	No	No	No	No	No	No	No	No	Yes
Storing com-posite service	N/A	Yes	No	No	N/A	No	No	No	Yes	Yes
Ranking com-posite services	No	No	User-defined criteria	QoS con-trol	N/A	No	No	No	Yes	Yes

Chapter 3

Overall Architecture

To address the shortcomings of existing service composition systems identified in Section 2.3.3, in this chapter, I describe an architecture that can make use of semantic-based service composition to dynamically build workflows without direct user involvement. My proposed architecture supports a variety of protocols and does not involve the end-user in specifying a composite service. The similarity of different pervasive environments in terms of type of devices and services, pattern of service usage, etc. suggests having a shared repository of predefined as well as on-the-fly generated workflows. Such a repository will increase re-usability of workflows and decrease the overhead of composition. It also offers the potential to assess the usefulness of compositions stored in the repository by tracking their use. My proposed architecture involves third-party SEs to keep such a shared repository. The SE uses a ranking scheme to decide which of the automatically generated composite services should be presented/offered to users. Composite service usage information, collected by the SE in the shared repository, from similar environments, is also used in my architecture to further filter out automatically generated workflows that are unlikely to be useful.

The presentation of my architecture in this chapter is high level and discusses only

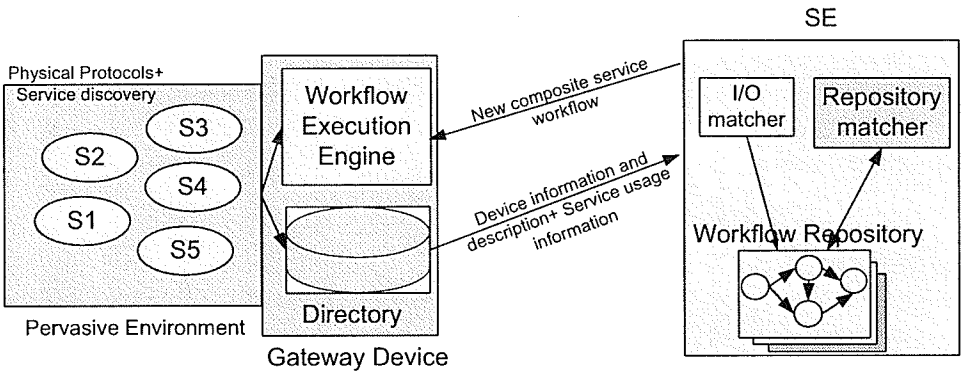


Figure 3.1: Overall architecture for service composition

the major components. I assume a pervasive environment in which devices are connected to each other (using wired or wireless technology) as well as to some sort of remotely managed GD which provides Internet access and hence, indirectly, support for composition via communication with an SE.

Figure 3.1 shows the overall architecture. Each device provides one or more services (S_n in the figure) which can be used by other devices, as part of composite services, or by users within a pervasive environment. Devices in the architecture may use different communication as well as service discovery protocols (e.g. Jini, UPnP, etc.) to advertise their services. The GD maintains a directory of all available services (The "Directory" in Figure 3.1) and is connected to one or more SEs through the Internet connection provided by the GD. SEs support service composition and may also provide remote services (e.g. health monitoring, entertainment, software upgrades, etc.) that are available in the pervasive environment.

The SEs perform service composition using repository-based (a.k.a. template-based)

and/or input/output-based (a.k.a. interface-based) matching, (as will be described in Section 3.1) based on device information sent by the GD. As shown in Figure 3.1, the GD sends newly added devices' descriptions and capabilities to the SE. The SE maintains a "shared" repository of pre-defined as well as automatically generated workflows used by the service composition modules (i.e. *I/O matcher and* Repository matcher in Figure 3.1).

It is assumed that all devices in a pervasive environment have built-in descriptions of their offered services, specified in OWL-S. Such descriptions, in a real world environment, could also be accessed by the SEs, via the Internet, on the device manufacturer's Web site based on a simple model number provided by the device. The implementation dependent part of each OWL-S description, (i.e. the "grounding") must be generated based on each device's supported communication protocol. This means that the GD, which runs the middleware, must create the service grounding dynamically and attach it to the service description, encoded in OWL-S.

Workflows in my architecture come in two forms: *abstract* and *concrete*. The descriptions of component services in an abstract workflow are used to discover concrete services (provided by a device or software) when such services are available in a pervasive environment. This type of workflow is maintained in the SE-side repository. Concrete workflows, on the other hand, are "instances" of abstract workflows and carry, in addition to abstract description of services (i.e. service "profile" in OWL-S), the implementation dependent part of services (i.e. "grounding" in OWL-S). When service descriptions are sent to the SE by the GD (whenever a new device is discovered), they are matched against the components of abstract workflows. If all the components of an abstract workflow are matched with descriptions of available concrete services, the concrete workflow, corresponding to the abstract one, is sent to the GD for deployment in the pervasive environment.

Given information about existing devices and services in a pervasive environment, the SE can also automatically discover new composite services (using input/output matching)

of potential interest to the users. The SE is expected to be connected to many different GDs and this provides the SE with the opportunity to share composite services among different pervasive environments. Reusing pre-existing compositions can be significantly more efficient than re-generating them. This is important when an SE "manages" many GDs.

When a new high-ranked composite service is found, the SE adds it to the repository and sends the description of the new composite service as a workflow to the GD. The *Workflow Execution Engine*, shown in Figure 3.1, is responsible for receiving and later deploying the workflows sent by the SEs. The workflow is already bound by the SE (using device information received from the GD), using its OWL-S grounding, to the available services/devices in the target pervasive environment. The workflow is thus ready to be executed by the GD. A GD may also, however, have to consider certain constraints (e.g. availability of protocols) during the eventual workflow execution. Such execution details, however, are beyond the scope of this thesis.

Deployment of composite services (for subsequent execution) in my architecture is handled in a unique way which makes it possible to integrate newly created composite services with available ones seamlessly. Each composite service is deployed in terms of all the available protocols (e.g. UPnP, Jini, etc) in a pervasive environment. To do this, a *wrapper* is dynamically created (by the Workflow Execution Engine) for each available protocol and for each new composite service when it is received by the GD. Each generated wrapper translates the composite service's functionality to one of the existing protocols in the pervasive environment. The wrapper service is first registered with the GD and then becomes available for use in the pervasive environment. When a composite service is accessed by a user or other services, usage information is sent to the SE by the GD. The SE collects statistics about composite service usage to identify "similar" pervasive environments in terms of their service usage patterns and to determine the "popularity" of

specific compositions so that the SE can discard less useful composite services accurately.

Although creating a wrapper for each available protocol in a given environment seems to be extra work, it integrates the new composite services without having to change any protocols or install any additional software. Furthermore, it can also be safely assumed that only a few protocols will likely be widely used in pervasive environments.

3.1 Service Composition Methods

Service composition at the SE side is done in an incremental fashion. In other words, each time an SE receives a new service (or device) description, it tries to find new composite services (using either or both of the following methods) that involve the newly added service and other pre-existing services within the pervasive environment.

- Repository-based service composition: In the repository-based approach, we use the pre-existing repository of useful composite service templates residing at the SE. When the SE receives a service description (provided by a newly discovered device) from a pervasive environment, it tries to find a match between the abstract description of a service in any of its templates and that of the received service. The *Repository matcher* in Figure 3.1 is responsible for doing this "repository matching". The match is *exact* if we can find a service from the repository where the number and type of its inputs and outputs are exactly the same as those of the received service. If we can find a subsumption relation between the inputs/outputs of the received service and a service from the repository, the match will be *partial*. (More specifically, if an input type of the received service is a superclass of the corresponding input type of a service from the repository and an output type of the received service is a subclass of the corresponding output type of service from the repository, a partial match will be detected.) If a match is either exact or partial, the SE will mark this service in the

Table 3.1: Comparison of different service composition methods

Method	Pros	Cons
Template-based	simple, fast, useful composition	limited to known components
Interface-based	simple, flexible	time consuming, requires ranking

template as "resolved". If all the services of a template are resolved based on services provided within the pervasive environment in question, then a composition has been found and the SE can send the *workflow instance* (concrete workflow) corresponding to the matched template to the GD. Otherwise, no match is found. Marking a service at the SE side indicates that this service is provided in a pervasive environment and can be used in creating a workflow instance.

- Input/output-based service composition: An SE tries to find a sequence of services, corresponding to those available in a pervasive environment, where the output(s) of each service are compatible (syntactically or semantically, given an ontology) with the input(s) of the next. The *I/O matcher* component in Figure 3.1 is responsible for doing interface-based matching. If such a sequence is found, the SE calculates a rank (reflecting quality) of this sequence based on a set of parameters and sends the generated workflow to the GD if the calculated rank is higher than a threshold value. Compositions (workflows) discovered by such I/O matching are then stored as templates (abstract workflows) in the shared repository to facilitate subsequent compositions using repository-based method. (The link between the I/O matcher and the workflow repository in Figure 3.1 shows this relationship.)

Each method of service composition has pros and cons that are summarized in Table 3.1. In some sense, the techniques complement one another, with each addressing inadequacies of the other. By using them together, template-based first, if possible, followed by interface-based, only if necessary, a powerful composition system results.

3.2 Ranking Composite Services

The number of composite services generated using I/O matching can be large. I use *abstract* and *concrete* rankings, corresponding to abstract and concrete workflows, to select the best possible workflow instances in a pervasive environment to deploy and execute. Abstract ranking is applied at the SE side, where abstract workflows are created using I/O matching, whereas concrete ranking is used by the GD when it receives a concrete workflow from an SE. Each ranking scheme, as will be described in Section 5.4.4, has its own set of criteria to calculate the rank of a composite service. In my prototype I only implemented and used abstract ranking to reduce the number of composite services sent to the GD.

The key challenge to making ranking effective is selecting a set of criteria for the rank function and the threshold value(s) as cut-off point(s) so that only compositions that are of end-users interest will be deployed. To select these values in my thesis, I reviewed the results of composing services whose input and output types were selected from a realistic ontology. Then, I selected and fine tuned the weights of different criteria in the ranking function (according to the structure of the type ontology, etc.) to cluster composite services into different categories with distinct rank values. Having these categories, it is not hard to determine appropriate threshold values to identify and then discard non-useful (low-ranked) compositions. The realistic domain ontology that I developed is presented and explained in Chapter 4.

3.3 Fault Handling in Composite Services

A composite service, which is supposed to be deployed in a pervasive environment, includes component services that can be either atomic or composite. A composite service may fail due to the failure of any of its components. The fault detection and tolerance mechanisms discussed in Section 2.1.3, for a service in general, can also be applied to a composite service's components.

In a pervasive environment where devices/services come and go frequently, the unavailability of a given service will affect all composite services having the failed service as one of their components. Detection of failed service components can be done using either heartbeat messages or a lease mechanism (see Section 2.1.3.1). Once a failed service is detected, we can either immediately make the involved composite services unavailable (due to the unavailability of the failed service) or find an alternative component (or composite) service. These approaches can either be done immediately after finding a failed service ("eager") or postponed until the composite service is actually invoked ("lazy").

To choose the most appropriate of the aforementioned methods in case of service failure, services in my system are classified into *mobile* and *stationary* depending on the type of device hosting the service. To distinguish mobile services from stationary ones, an extra tag is added to the OWL-S description of each service (see Figure 3.2). This tag identifies the type (mobile or stationary) of the device hosting the service. If a composite service has at least one mobile service, it will be marked as mobile as well.

If a mobile component service becomes unavailable (assuming that unavailability is detected using heartbeat messages), we wait until the composite service is actually invoked hoping the mobile component service will become available again. On the other hand, the unavailability of a stationary service can imply a permanent problem (at least a longer lasting one than a mobile component) so the SE is notified about removal of this service.

```
<profile:Profile rdf:about="#Service1Profile">
  <profile:serviceName>Service1Service</profile:serviceName>
  <profile:hasInput rdf:resource="#input1"/>
  <profile:hasOutput rdf:resource="#output1"/>
  ...
<hostDevice:serviceType>Mobile</hostDevice:serviceType>
</profile:Profile>
```

Figure 3.2: Addition of the new `serviceType` tag to the service description

As described earlier, each SE maintains a repository of predefined abstract workflows (i.e. potential composite services) and marks components of workflows as "resolved" upon receiving description of matching concrete service from a given pervasive environment. Each SE typically manages many pervasive environments, thus the marking of component services is done on a per-environment basis. In other words, the SE will only send a workflow to a GD if all the component services are marked as "resolved" and corresponding concrete services belong to a single environment.

Marking a service, as mentioned earlier, indicates the use of such a service in a composite service for some pervasive environment. To deal with service removal/failure in a pervasive environment, the inverse processing must be performed. Thus, upon detecting a service failure for a fixed device, the GD must send the description of the failed service to the SE so it can *unmark* this component service in all abstract workflows having the description. Unmarking a component service within a workflow does not have any effect on the workflow structure. It just prevents sending workflows including a component service

whose abstract description matches with the description of the failed service to the environment in which service failure occurred. The corresponding abstract workflow remains in the repository and can be used in matching and to send a corresponding workflow to another pervasive environment if all its component services are available in the environment.

Not sending any workflows including failed services only partially solves the problem of handling failed services. We also need to deal with those composite services that have already been sent to a GD and which involve a failed component service. To avoid losing pre-deployed services in a given pervasive environment, we need to be able to find an alternative component (or composite) service.

3.3.1 Selecting an Alternate Service

The GD, which is responsible for deploying composite services, detects the unavailability of a service at execution time. The process of handling a failure, as we mentioned earlier, can start either right after detecting the failure or at the time of actual execution of a composite service using the failed component service. With either approach, the GD can either generate an error message and show it to the requester or select an alternate service (composite or atomic). Selecting an alternate service is clearly preferable and can be done by replacing either the whole composite service with a new one (given that such a backup service is available) or by replacing only the unavailable component service. In some cases, replacing only the unavailable component service can save computation by not re-invoking already executing component services. However, finding another component service exactly matching the failed one may not be simple. Also, the result(s) of previously executed services must be saved using a mechanism such as checkpointing so they can be used to resume the execution of failed composite service.

To be able to discover alternate services, extra information must be kept about the available services in the OSGi framework in terms of their input(s) and output(s). A challenge to doing this is that the gateway device typically is an embedded device having limited resources so we must keep the overhead of finding new alternative services to a minimum. This can be achieved by using only the exact matching of input and output types without considering semantic matching and can be implemented using a list to map each service (atomic or composite), based on its input and output types (available from the OWL-S description of the service), to an entry in the OSGi directory. This list can be easily updated on the arrival/removal of a service. Having this mapping enables us to cheaply find other services (if such services exist) which can act as a replacement for a failed service.

If a replacement for a component service cannot be found, the list can be searched to find another service to replace the whole composite service. If a replacement for a composite service can not be found, an error message must be sent to the requester of the composite service. Handling a failure on the GD by selecting an alternate composite service is not provided in the current implementation of the prototype.

3.4 Prototype Implementation Overview

In my prototype (see Figure 3.3), I have implemented a GD running the Oscar [3] implementation of the OSGi [17] framework. My prototype currently supports a few real as well as a number of emulated UPnP devices and Jini services. All services are registered using OSGi in the GD. A separate program (that can run on the same machine or a separate computer) acts as an SE which is responsible for receiving service descriptions from the GD, retrieving compositions from the shared repository and/or creating new composite service workflows using input/output matching, and sending them (if the SE finds high-ranked composite services) back to the GD. As mentioned earlier, each service carries its

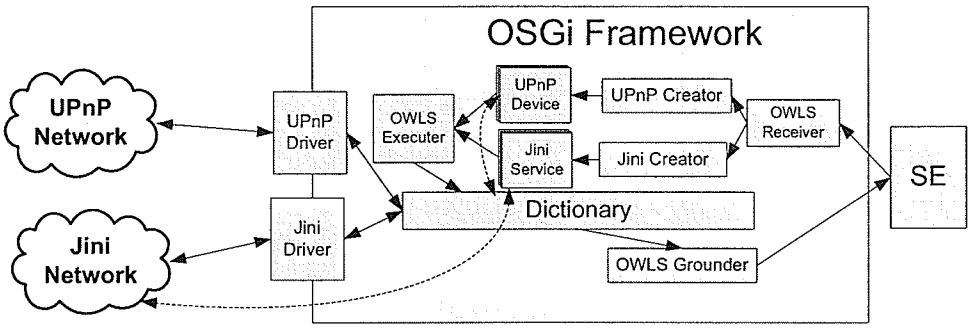


Figure 3.3: Components of the prototype implementation

own description encoded in OWL-S.

To take advantage of the interoperability provided by OSGi, I have implemented OSGi groundings to bind service descriptions to their implementations in OSGi. Using OSGi as a grounding mechanism allows my system to support *heterogeneous* composite services. For example, one component of a composite service can be selected from the available UPnP services while another can be selected from, for example, the available Jini services. The grounding part of a service is created on the fly when the service is registered with OSGi. Once a service is registered in the GD (using one of the Jini or UPnP drivers shown in the figure), I extract its semantic specification (encoded in OWL-S) as well as information about the number of inputs and the types of each input. Then, I use the OWL-S API [76] to create a grounding using the extracted information. The "OWLS Grounder" module, shown in Figure 3.3, creates the grounding part of the OWL-S description for a registered service. Once the grounding is created, I append the grounding to the semantic description of the service.

Once the service grounding is created, the whole OWL-S file, including the service "profile", "process", and "grounding", is sent to the SE (i.e. the program acting as the

SE in my prototype). The SE then uses the previously described composition techniques, both template-based and interface-based, to find useful composite service workflow(s). If such workflow(s) exist, the SE will send them, encoded in OWL-S, back to the GD. The "OWLS receiver" module in Figure 3.3, receives the OWL-S description from the SE and subsequently calls the appropriate code modules to create Jini and/or UPnP wrappers based on the abstract description of the received composite service.

The GD deploys all received composite services by creating wrappers for them. By creating a wrapper and registering it within OSGi, the other devices in a pervasive environment will be notified of the existence of the new service. After that, devices can interact with the new composite service(s) just like any other service. The invocation of a new composite service goes through one of the Jini or UPnP drivers to the OSGi framework and the "OWLS Executer" bundle performs the required operations (i.e. calling component services) to return the result of calling a composite service to the caller. The "OWLS Executer" bundle also sends service usage information to the SE to be stored in the shared repository.

3.5 Summary

In this chapter I provided an overview of my architecture for service composition in a pervasive environment. The architecture does not involve the end-user in specifying a composite service but, instead, offers a list of highly ranked and hence hopefully useful composite services to the end-user to choose from. Pervasive devices are usually resource-limited and their corresponding environments are similar in terms of available devices, usage pattern of services, etc. Thus, my architecture involves an SE in the service composition process to provide not only enough compute power required for semantic matching but also a repository to store composite services that can be shared among different pervasive environments.

The SE uses input/output matching to create new composite services that do not exist in the repository. Input/output-based service composition can potentially generate a large number of composite services and only a subset of them are likely to be of end-user interest. Thus, the SE uses a ranking method to prune composite services that are less likely to be useful in a given pervasive environment. Information about the usage of a composite service in similar environments is used to adjust the computed rank of automatically generated composite services.

Chapter 4

Designing a Domain Ontology

The semantic matching of services' input and output types requires a common definition of types. Such a definition is usually abstract (to make it independent from a programming language and run-time platform) and specific to a domain. For example, the input type of a "Print Service" might be "PostScript File" which should be well-defined and its relationship to other types of a file must be identified to permit semantic comparisons. One generally accepted approach to provide definitions about different concepts of a domain and to present relationships among them is using an *ontology*. An ontology, as will be used in the next chapter, is also related to ranking generated composite services. Different criteria for ranking and corresponding weights are selected, in part, based on the structure of the domain ontology. A detailed and precise ontology results in more accurate composition and in better ranking of services.

An ontology [41] "is a formal, explicit specification of a shared conceptualization. Conceptualization refers to an abstract model of phenomena in the world by having identified the relevant concepts of those phenomena. "Explicit" means that the type of concepts used, and the constraints on their use are explicitly defined. "Formal" refers to the fact that the

ontology should be machine readable. "Shared" reflects that ontology should capture consensual knowledge accepted by the communities". An ontology interweaves human and computer understanding of terms and relations [35]. An example of such a relationship is the superconcept-subconcept relationship (a.k.a. "is-a" relationship). Such relationships are easy to understand by humans and, since they are formally defined, a computer can capture/model the relationships that a human perceives.

There have been many ontologies developed for different purposes in different areas. However, the generality (i.e. scope) of these ontologies as well as their expressiveness (i.e. the level of details that they can provide) are not the same [35]. To be able to assess my service composition system, I need one or more domain ontologies suitable for home area networks and other local interaction environments. The developed ontologies must capture the possible types used to describe a service's inputs and outputs, and their relationships. For example, an ontology dealing with personal entertainment services should demonstrate that MP3, WMA, etc. are different types of audio files. To express the desired ontology I use OWL which can be easily integrated with OWL-S for service description. The Amigo project [37] has already defined several ontologies mostly for describing and categorizing *devices* in pervasive environments such as a home. However, my system requires a type ontology to be able to match the input and output types of services semantically. Thus, the available ontologies from the Amigo project cannot be directly used in my system to describe the different input and output types.

In the rest of this chapter I focus on two main issues related to the design of a type ontology: representing the types existing in a domain, and considering the different relationships (other than just "is-a" relationship) among identified types.

4.1 Modeling a Type Ontology

My primary test environment is a home area network with a variety of devices of different types. Each device offers one or more services. The functionality of a service is partially described by its input and output types. Inputs and outputs must take their types from a "domain ontology" to make semantic matching possible. As an example, consider part of a simple type ontology describing audio types as shown in Figure 4.1. (My complete ontology is depicted in Appendix B.) In the figure, numbers following different MP3 types represent bit rate and sampling rate used in a particular MP3 file respectively. Similar to the class concept in object oriented programming languages, a class in a type hierarchy can be either abstract or concrete. Abstract classes cannot have instances and should not appear as input or output types of a service unless the service supports *all* the concrete subclasses of that type as its input or output types. Furthermore, concrete classes can have instances and instances of a superclass are not necessarily instances of its subclasses. For example, we might have an "Address" class describing the general concept of an address. This class can have a subclass called "AddressWithPostalCode". This subclass has a property such as "postalCode" which its parent class does not. Thus, if we have an instance of address without postal code, it will be considered an instance of the Address class not the AddressWithPostalCode class.

Using a type hierarchy reflecting "is-a" relations, however, is not as simple when applied to automating service composition as it can be when used in programming language type systems. Using a general class such as MP3 as an input or output type can cause an incompatibility problem at run-time. Specifically, the MP3 type intuitively has properties such as bit rate and sampling rate that distinguish different kinds of MP3 files. Some MP3 playing services (offered by devices or software) may not be able to play all such kinds of MP3 files. Not considering such properties at matching time can therefore result in an

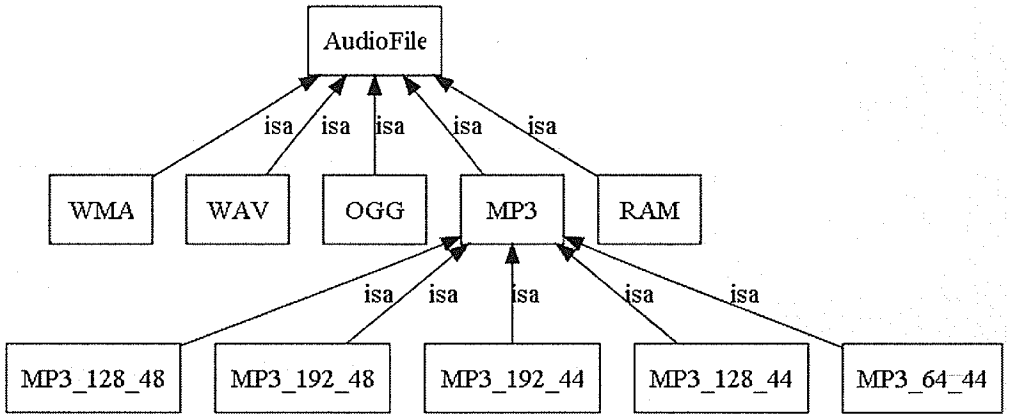


Figure 4.1: Part of a type ontology

apparently legal composition which will fail at run-time due to having an incompatible input type. This is highly undesirable in a setting where users are typically nontechnical in nature.

Unfortunately, there is no straightforward and unique way of representing and later using properties at matching time. Trying to model different types of an MP3 file as shown in Figure 4.2 violates "is-a" relationship that we have among other types. Since, in the figure, one cannot say that MP3_128 "is-a" or "is a kind of" MP3_192.

Generally speaking, we can either use a generic type with properties or different properties can be modeled using separate subtypes. Without loss of generality, I consider the MP3 type as an example to explain these two options. The same argument is, of course, valid for other similar types (e.g. video files, ...).

As an example and to assess the capabilities of a *personal* MP3 player in playing MP3 files, I arbitrarily chose a Samsung YP-P2JCB portable MP3/Video player as a sample device. Based on its user manual, it is capable of playing MP3 files with bit rates between 8 and 320 kbps and a sampling rate of 22 to 48 KHz. Commonly used bit rates in MP3

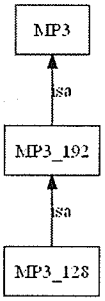


Figure 4.2: Modeling MP3 files similar to this hierarchy violates the "is-a" relationship

encoding are 8, 16, 32, 64, 96, 128, 160, 192, 224, 256, and 320 kbps. (The most common one is 128 kbps.) The sampling rate can have values such as 11.025, 22.05, 32.0, 44.05, 44.1, 47.25, and 48 KHz.

Having this information, we can categorize MP3 files using one of the following two methods: 1) every possible combination of properties appears as a separate type (similar to Figure 4.1), and 2) there is a single generic type with properties for bit rate and for sampling rate.

4.1.1 Properties as Types

If the values of a property can be discretized, we can enumerate all of the possible values as different types. This must only be done for *important* properties of a type. Important properties are those properties whose values have an impact on the successful execution of the generated composite service. For example, two important properties of an MP3 file (i.e. bit rate and sampling rate) might be enumerated using commonly used values. Thus, for our sample device, we would have 77 (11 (number of bit rates) * 7 (number of sampling rates)) different types covering all realistic/implemented combinations.

Using this approach the example MP3 player must advertise 77 different versions of the PlayMP3 service, one for each of the different types. This argument also holds for other media types supported by the example MP3 player.

The advantage of this approach is the simplicity (and potentially enhanced accuracy) of composition. At composition time, the inputs and outputs of services can be simply compared based on the pre-agreed types and there is no need to deal with separate properties during composition. It is also guaranteed that no composition will fail at execution time due to incompatible types. However, possible disadvantages of this approach are:

- The number of services advertised by each device will increase as the number of properties of a type increases. This may be tedious for device manufacturers and will increase matching overhead due to the need to deal with so many services versions.
- Limited applicability. If we cannot discretize the values of a type, all the legal values cannot be enumerated and matching between two services producing/consuming such types will be hard or even impossible. While clearly an issue, this problem does not appear to be common in practice.

4.1.2 Generic Types with Properties

Using properties there will be a single generic type for MP3 with different properties such as bit rate and sampling rate. Therefore, an MP3 player will announce only one playing service accepting MP3 type with specified properties having constrained values as its input. Possible advantages of this approach are:

- The number of advertised services by a device is much lower compared to the previous approach. This simplifies the advertising of a service by a device, may be more convenient for manufacturers to specify and will require fewer but more complex matching operations.

- Properties with continuous values can be presented in this method. A property such as "length" can be given an upper and lower bound and services are free to choose any arbitrary values from the pre-defined range. Logical operators (i.e. >, >=, <, etc.) can be used to compare two properties with continuous values.

The major disadvantage of using properties is the current lack of a standard way to describe and later use them. A service described in OWL-S can include preconditions and post-conditions to check/change properties. However, there is no standard way of describing pre/post-conditions. There are also other issues such as handling missing properties and properties without values, which add to the difficulty of using this approach.

4.1.3 The Selected Modeling Strategy

Considering the complexity of handling properties at matching time and the importance of these properties in composition, I chose to model important properties (i.e. properties such as bit rate and sampling rate of an MP3 type) having separate types.

To remove or lessen the burden of creating many services offered by each device, a single service using generic types (higher level classes in the type hierarchy) could be described (by the device manufacturer for example) and a set of services whose inputs and/or outputs are from each subclass of the defined type could then be generated automatically based on the structure of the type ontology. For example, the manufacturer of an MP3 player which is capable of playing any type of MP3 file (based on the agreed-to type ontology) can describe a service accepting an input of the generic type MP3 and services with inputs taken from each subclass of the MP3 type would be generated. This approach solves the *evolution* problem associated with using generic types. For example, if the MP3 type evolves over time and new subtypes are introduced (e.g. a higher encoding rate is added as technology improves), existing devices will still work since they have not announced the

generic MP3 type. On the other hand, device manufacturers can freely incorporate new subtypes in the services offered by their new products.

4.1.4 Using the Selected Type Ontology

Given the type ontology subgraph shown in Figure 4.1 with distinct subclasses for important properties of the generic MP3 type, consider an MP3 player capable of playing MP3 files with 128 kbps bit rate as well as recording audio as MP3 files with 64 kbps bit rate (assuming 44 kHz sampling rate in both cases). This player would announce the following services (service names are shown in **bold** face):

$$MP3\_128\_44 \rightarrow \mathbf{PlayMP3\_128\_44} \rightarrow none$$
$$none \rightarrow \mathbf{RecordMP3\_64\_44} \rightarrow MP3\_64\_44$$

Now, consider a service, provided by software, capable of playing all types of MP3 files as well as converting them to wave files. This software might announce the following services:

$$MP3\_64\_44 \rightarrow \mathbf{PlayMP3\_64\_44} \rightarrow none$$
$$MP3\_64\_44 \rightarrow \mathbf{StreamMP3\_64\_44} \rightarrow wave$$
$$MP3\_128\_44 \rightarrow \mathbf{PlayMP3\_128\_44} \rightarrow none$$
$$MP3\_128\_44 \rightarrow \mathbf{StreamMP3\_128\_44} \rightarrow wave$$
$$MP3\_192\_44 \rightarrow \mathbf{PlayMP3\_192\_44} \rightarrow none$$
$$MP3\_192\_44 \rightarrow \mathbf{StreamMP3\_192\_44} \rightarrow wave$$
$$\ldots$$

If there is some other software capable of encoding wave files into MP3s (any type of MP3), it would announce services such as:

$$wave \rightarrow \textbf{EncodeMP3\_64\_44} \rightarrow MP3\_64\_44$$
$$wave \rightarrow \textbf{EncodeMP3\_128\_44} \rightarrow MP3\_128\_44$$
$$wave \rightarrow \textbf{EncodeMP3\_192\_44} \rightarrow MP3\_192\_44$$
$$\ldots$$

Having all these services would enables the creation of a composite service such as:

$$\textbf{RecordMP3\_64\_44} \rightarrow \textbf{StreamMP3\_64\_44} \rightarrow \textbf{EncodeMP3\_128\_44} \rightarrow \textbf{PlayMP3\_128\_44}$$

4.2 Ontological Siblings and Matching

There may be composable services whose input and output types are siblings of a generic type in the type hierarchy. In such cases, this reflects a relationship that might be useful in semantic matching. For example, MP3 and WMA are two audio file types which are subclasses of the generic type AudioFile in Figure 4.1. According to the normal definition of semantic matching, there is no match between a service with input type WMA and a service with output type MP3 since they are not related by an "is-a" relationship. In this case, however, they are clearly related and it would be logical to consider replacing a WMA device with a similar MP3 device in some situations. To accomplish this, some intermediate *glue* services are, of course, needed to convert between the types. This is complicated by the fact that not all sibling types are related, so simply extending the "is-a" relationship used in semantic matching is unsafe. Figure 4.3 shows part of an ontology describing different

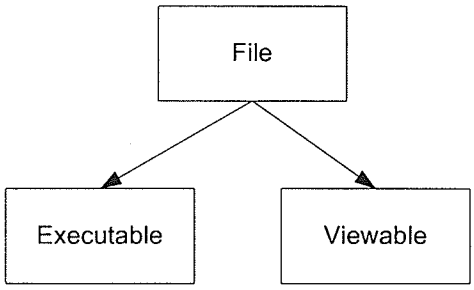


Figure 4.3: Part of a type ontology in which siblings do not have any relationship

file types. Although "Executable" and "Viewable" types are siblings, it does not imply any relationship among them.

The penalty of using such glue services will be reflected in having longer composite services. Having a hierarchical type ontology, however, does enable the domain experts maintaining the ontology and using it to define the services offered by specific devices to easily recognize related sibling types and provide appropriate glue services as needed to support compositions that are foreseen to be useful. In my prototype, glue services can be offered as OSGi bundles and deployed remotely on gateway devices.

Chapter 5

Implementation Details

In this chapter I explain the details of describing a service in OWL-S, integrating (as an example) a UPnP device and service description, and most importantly, the composition process taking place at the SE side. I also provide a basic complexity analysis of the process of finding a sequence of services at the SE side.

5.1 Service Description

OWL-S is a language for the semantic description of services. It uses OWL, an ontology language, to describe services input and output types. OWL-S is also capable of describing composite services as workflows. Services, provided by different devices, are described in OWL-S using three different aspects: service profile, service process, and service grounding. A service profile describes a service in terms of its inputs, outputs, pre-conditions, and effects. The inputs and outputs of a service have types which come from an ontology such as described in Appendix B. These types are used for subsequent semantic matching. A service process explains how a service works. The service process for a composite service shows how different components interact with each other. Finally, a service grounding

provides required information about how to access the service. Service grounding is implementation dependent and in my prototype system, I have implemented an OSGi-based service grounding for OWL-S descriptions.

Although it is not practical to create a unique standard ontology to be used across all possible pervasive applications, there are a few existing ontologies that can be extended to accommodate different applications' requirements. The Simple Ontology for Ubiquitous and Pervasive Application (SOUPA) [28], for example, is an ontology developed to support knowledge sharing and interoperability in ubiquitous and pervasive systems. This ontology can help system developers to quickly build ontology-driven applications without spending too much time on building their own ontologies from scratch.

SOUPA is encoded in OWL and comprises two distinct but related ontologies: the SOUPA Core and the SOUPA Extension. The SOUPA Core includes common vocabularies (e.g. time, event, person, etc.) that are universal among different pervasive applications. Extended from the SOUPA Core, the SOUPA Extension defines additional vocabularies (e.g. meeting and scheduling, location, etc.) for specific applications. The SOUPA Extension also demonstrates how a new ontology can be defined by extending the SOUPA Core ontology.

To implement my prototype, I have created a simple yet "realistic" domain ontology (as described in Chapter 4 and shown pictorially in Appendix B). In a real implementation of a service composition system, more general ontologies (e.g. SOUPA) could also be used. Different ontologies from different sources, however, are not necessarily compatible. The process of merging different ontologies is, itself, an interesting research topic [53] but one that is outside of the scope of this thesis.

For the rest of this chapter I assume that a multitude of services may exist in a given pervasive environment of interest. I further assume that their descriptions are available in OWL-S. In a typical, real-world pervasive environment a large number of service-offering

devices will be available ranging from the powerful and complex (e.g. a home PC) to the simple and small (e.g. sensors). The services offered are, naturally, device specific and must be composed correctly to provide useful functions. Let us consider two simple services, `GetStatus` and `SetPower`, that might be offered by two devices, namely a computer-controlled (binary) switch and a TV. The TV might advertise its service(s) using the UPnP protocol and the software switch might use Jini. Using these two services we can see how the service composition process creates a sequence combining the services and how a new composite service is deployed. While this example and others are focused on home networks, my method for composition is general and can be equally well used in other local pervasive environments such as smart meeting rooms, airport lounges, etc.

To start the composition process, the semantic description of a service encoded in OWL-S is needed. Figure 5.1 shows a description of the `GetStatus` service which reads the status of a binary switch. (The grounding part is implementation dependent and is not shown in detail in the figure. I will describe it later in this chapter). Recall that OWL-S describes a service based on three components: profile, process, and grounding. The `service:presents` tag creates the link between a service and its profile. The description of a profile starts with the `profile:Profile` tag in the OWL-S description. The `service:describedBy` tag specifies the process associated with this service. A process can be atomic, simple, or composite. A simple process is not directly callable and serves only as an abstraction for an atomic or composite service. In Figure 5.1 the description of an atomic process starts with the `process:AtomicProcess` tag. The `service:supports` tag links the service with its grounding. The OSGi grounding of a service is described using the `grounding:OSGiGrounding` tag. In addition to these three components of a service, the input(s) and output(s) of a service can be described in the OWL-S description as well. In Figure 5.1 the only output of the `GetStatus` service is described using the `process:Output` tag.

The binary switch can be in the "On" or "Off" state. As can be seen from the description in Figure 5.1, the output of the `GetStatus` service is of type `Status` which is defined as an OWL class in our domain ontology. Similarly, there will be a description for the `SetPower` service which accepts one input and, based on the input value, turns the TV on and off. Having the description of both `GetStatus` and `SetPower` services in OWL-S, one can see that these two services are input/output compatible and composing them results in a computer-based on/off control for the TV.

I assume that each device carries a link to its service's description. If a device is not capable of carrying this link, the description can be downloaded by looking up the device's model number on the Internet. A UPnP device, normally, has a set of built-in attributes (e.g. a descriptive name, manufacturer's name, etc.) that are announced at discovery time. These attributes provide information about the device. This provides a mechanism by which UPnP devices can provide a link to their services' descriptions. I assume that UPnP devices do carry links to their service descriptions, encoded in OWL-S, in one of these attributes. Alternatively, if space on the device is limited, the actual description can be kept on the device manufacturer's Web site and a URL to the Web site provided by the relevant attribute. If a UPnP device has more than a single service, the links to different services can be provided in a single string separated by a pre-agreed on delimiter.

Figure 5.2 shows a component view of the implemented bundles within my OSGi prototype. The lines in this figure show dependencies between components and the arrows show the direction of each dependency. Once a UPnP device, for example, becomes available in a pervasive environment, it will be detected and subsequently registered within OSGi using the UPnP driver bundle. Then, a bundle, OWLS Grounder[1], creates the grounding part for each service provided by the UPnP device. (Figure 5.3 shows a sequence diagram for this operation.) The grounding, created in this way, is specific to OSGi and has

[1]The components of my prototype, including the OWLS Grounder, were presented in Figure 3.3

```
<service:Service rdf:ID="GetStatusService">
  <service:presents rdf:resource="#GetStatusProfile" />
  <service:describedBy rdf:resource="#GetStatusProcess" />
  <service:supports rdf:resource="#GetStatusGrounding" />
</service:Service>

<profile:Profile rdf:ID="GetStatusProfile">
  <service:presentedBy rdf:resource="#GetStatusService"/>
  <profile:serviceName xml:lang="en">GetStatusService
  </profile:serviceName>
  <profile:hasOutput rdf:resource="#RetTargetValue"/>
</profile:Profile>

<process:AtomicProcess rdf:ID="GetStatusProcess">
  <process:hasOutput rdf:resource="#RetTargetValue"/>
  <service:describes rdf:resource="#GetStatusService"/>
</process:AtomicProcess>

<process:Output rdf:ID="RetTargetValue">
  <process:parameterType rdf:datatype=
    "http://www.w3.org/2001/XMLSchema#anyURI">
    http://www.somedomain.org/DomainOnt.owl#Status
  </process:parameterType>
</process:Output>

<grounding:OSGiGrounding rdf:ID="GetStatusGrounding">
  <service:supportedBy rdf:resource="#GetStatusService"/>
  ...
</grounding:OSGiGrounding>
```

Figure 5.1: The description of the `GetStatus` service encoded in OWL-S

information about how to access the registered service within OSGi. Figure 5.4 shows the OSGi-specific grounding for the `GetStatus` service. The grounding, represented by `O.G:OSGiAtomicProcessGrounding`, specifies the type of input(s) and output(s) of

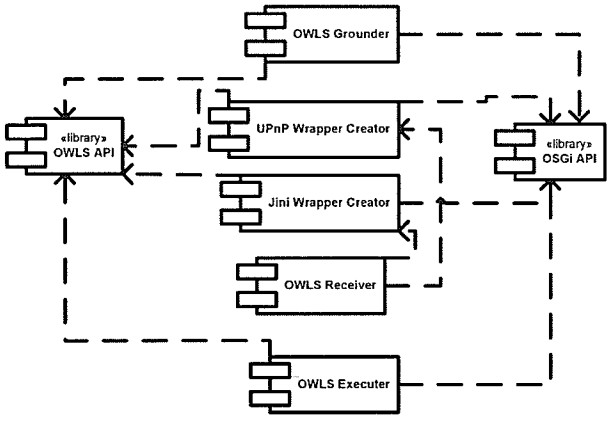


Figure 5.2: Component view of implemented bundles within the OSGi framework

the service based on Java's primitive data types. The `O.G:osgiClass` defines the class name by which the service is accessible within the OSGi framework. The service itself is described using the `O.G:osgiService` tag. The output of this service is described using the `O.G:osgiServiceOutput` tag. (This is similar to the Java grounding which is available in the OWL-S API [76].) The grounding is attached to the service description, provided with the device, and the whole description (service profile, process and grounding) is sent to the SE for matching and composition operations. (The "Request Receiver" component shown in Figure 5.3 represents the SE side receiving the service description from some pervasive environment.)

5.2 Semantic Matching

As described earlier, service matching is done at the SE side for a number of reasons including:

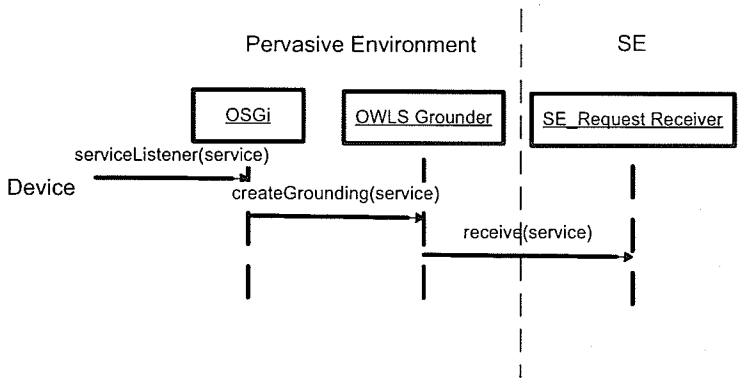


Figure 5.3: Sequence diagram for creating an OSGi grounding for an advertised service

```
<O.G:OSGiAtomicProcessGrounding rdf:ID="GetStatusProcessGrounding">
  <grounding:owlsProcess rdf:resource="#GetStatusProcess" />
  <O.G:osgiClass>org.osgi.service.upnp.UPnPDevice</O.G:osgiClass>
  <O.G:osgiService>GetStatus</O.G:osgiService>
  <O.G:osgiServiceOutput>
    <O.G:osgiVariable rdf:ID="Out">
      <O.G:javaType>java.lang.Boolean</O.G:javaType>
      <O.G:owlsParameter rdf:resource="#RetTargetValue" />
    </O.G:osgiVariable>
  </O.G:osgiServiceOutput>
</O.G:OSGiAtomicProcessGrounding>
```

Figure 5.4: OSGi-specific grounding for the `GetStatus` service

- Performing fully automated service composition without user involvement,
- Removing the overhead of service matching from resource limited pervasive devices,
- Enabling the (re)use of information between different environments using a shared

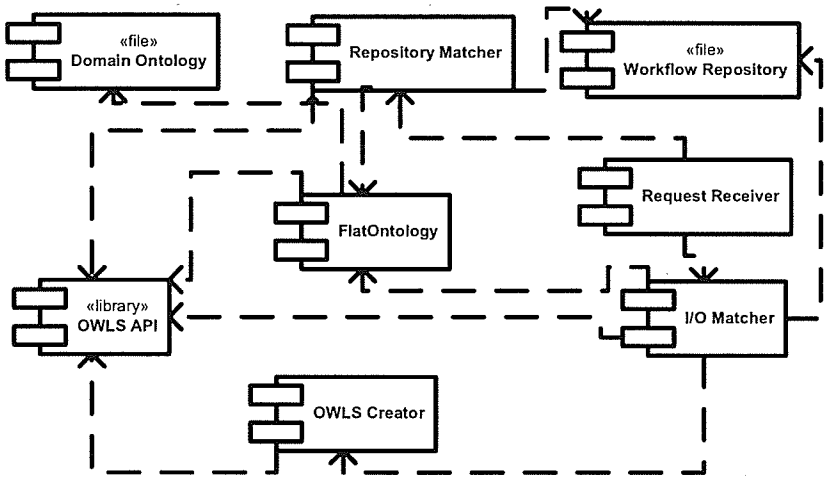


Figure 5.5: Component view of the SE side code used to match a composite service and create a workflow corresponding to it for a given pervasive environment

repository of composite services, and

- Identifying "similar" pervasive environments based on their service usage information.

Figure 5.5 shows the software components that have been implemented as part of the SE side code. This figure shows which components interact with each other and the direction of data flow.

Service composition done by the SE could, in general, be done using either or both of the repository-based and I/O-based matching schemes. In this section I explain the implementation details of each scheme and the challenges that I faced. Before going into the details of each scheme, I show the overall sequence of executed operations used to

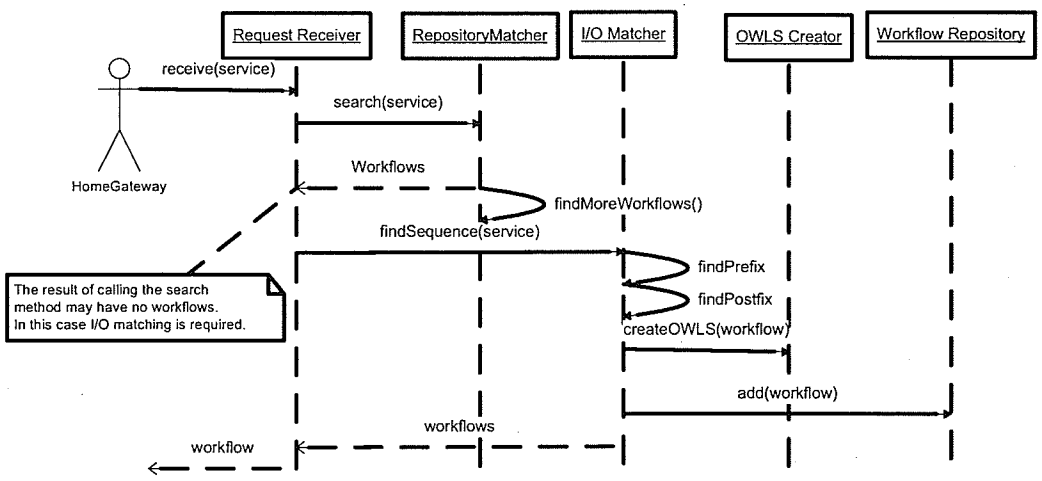


Figure 5.6: Sequence diagram for creating a composite service on the SE side

create a composite service in Figure 5.6. The `findMoreWorkFlows()` method in Figure 5.6 checks the workflows returned from the repository search to find further compositions whenever possible. In this figure, the `findSequence(service)` method is called if the repository search returns no workflows, otherwise no I/O-based matching is done and the corresponding overhead is avoided. Two methods, `findPrefix` and `findPostfix`, are called to find all possible composite services using I/O matching. These methods are described in Section 5.4.1.

5.3 Repository-based Service Matching

Workflows describing composite services are represented using OWL-S. Since it is useful to be able to share workflows between different pervasive environments, the workflows cannot simply be stored on participating computing devices in the pervasive environments. Instead, the workflows must be stored by a third party at a generally accessible location.

Such workflows are stored in a repository maintained by the service enabler.

Abstract workflows stored in the repository specify the relations between different abstract component services in a composite service. Having abstract/generic descriptions of services (described by OWL-S profiles) allows us to match newly discovered services with services in existing workflows. Once all the service descriptions in an abstract workflow have been matched with the description of service instances for a given pervasive environment, an instance of the abstract workflow (i.e. a concrete workflow) can be created for that environment.

Consider the example described in Chapter 1, where a new game console is added to a home network. The new game console will advertise its services such as "play video", "display picture", etc. which will be sent to the SE where they can be matched against the abstract descriptions of component services in repository workflows. If I assume an abstract workflow such as "capture image+display picture" is available in the shared repository, its component services might be matched with the description of a service coming from a digital camera (i.e. "capture image" producing an output of type JPG from the type ontology) and the new game console (i.e. "display picture" taking an input of type JPG and producing an output of type SDVSignal from the type ontology). After matching descriptions of these services, a concrete workflow will be ready to be deployed in the home.

Different devices will become available at different times and most likely will be provided by different vendors. Thus, no assumption can be made about the names of services and used to find matching services. Semantic descriptions of services (based on the common ontology) are required. This allows matching using only the input and output types of two services (independent of their names) to find matches.

All services as well as composite service workflows are described in OWL-S using the common domain ontology. Although OWL-S can describe a service in terms of its inputs,

outputs, pre-conditions, and post-conditions, I only use the description of input(s) and output(s) for matching purposes. I chose to do this because pre-conditions and post-conditions do not have a widely accepted standard notation for representation. More importantly, using pre/post-conditions increases the complexity of composition which means longer execution times for composition.

My matching algorithm must match a newly announced service, S, within a pervasive environment with *all* workflows in which S can participate. To find such workflows, I compare the inputs and outputs of the newly received service from the GD with the inputs and outputs of all services stored in the repository. If a service in the repository exactly matches with S (input and output types of the two services exactly match), S is marked in all workflows involving it as being *resolved* for the pervasive environment in question. If no service in the repository exactly matches the received service, I then try to find a *similar* service.

To find a similar service, I use the available semantic information associated with a service's inputs and outputs to do partial matching [63]. Specifically, I try to find a service, S_i, whose inputs are either subclasses or the same as the inputs of S and whose outputs are either super-classes or the same as the outputs of S. Services in the repository having subclass/superclass relations with the received service will be marked as resolved in the corresponding workflows for the pervasive environment in question. During such marking of services in the workflows, if all the services used in any given workflow are marked as resolved, that workflow is ready to be specialized to the available devices and deployed in the pervasive environment in question. Once a workflow is deployed, the workflow description in the repository is returned to its original state by removing all markings. Pseudocode for this operation is provided in the next section.

Once a matched workflow is returned to its original state, a *deployed* flag, associated with the workflow, is set indicating that the workflow has been deployed at least once. Some

time later, if a new service, from the same environment, becomes available that matches any of the services in the already deployed workflow, that workflow can be immediately re-deployed using the new service instance without having to match instances for all the other services in the workflow. This speeds the overall matching process and also decreases its overhead. The "resolved" and "deployed" tags are associated with the location of a service so that similar services from different pervasive environments do not interfere. For example, if a workflow has two services S_1 and S_2 and the SE received S_1 from GD A and S_2 from GD B, the SE will send the workflow to neither of the GDs.

My method of marking assumes serial processing in which only one process/thread can receive a new service and mark it in workflows at a time. If multiple processes marking concurrently (to handle very frequent service arrivals corresponding, for example, to the presence of many mobile devices) is required this method needs to be modified slightly. For example, to make sure that component services of a composite service are safely marked concurrently, a locking mechanism, used by the marking process at the SE side, can be used to guarantee the exclusive access to a service for marking. The inherent problems associated with the locking mechanism (e.g. deadlock) must be handled by the marking process accordingly.

In my prototype, the repository search is done first and if workflow(s) are available to be sent to a pervasive environment, no input/output matching is performed to find other possible compositions. However, this approach may miss some compositions that might be possible by composing a workflow from the repository with a newly received service whose description is not in the repository. Let us assume a composite service workflow comprised of "Scan to JPG" (with the output type JPG) and "JPG to PostScript" (with the input type JPG and the output type PostScript) services. If the description of a new "Print PostScript" (with the input type PostScript) service is received by the SE, it will not match the mentioned workflow. However, it is possible to extend the workflow by attaching the

newly received service at the end of it (by sending the output of "JPG to PostScript" to "Print PostScript").

To address this issue, I could use each of the workflows that is going to be deployed in a given pervasive environment to build new composite services using the I/O matching techniques described in Section 5.4. Building new services by involving already existing workflows, can be done in a "lazy" fashion, in the background, to maintain acceptable overall system performance.

5.3.1 Implementing Repository-based Matching in the Prototype

The repository was implemented using a hash table whose key is a combination of the input and output URIs of a service. The order of inputs and outputs in two service descriptions, one received from a pervasive environment and the other stored in the repository, can be different. To solve this ordering problem among inputs and outputs, I sort the input and output URIs separately (i.e. inputs are sorted together and outputs together) before combining and using them as a hash key.

Let us assume there is a service S_r in the repository with two inputs `Audio`, `Video` and an output `AV`. Also assume there is a service S_p, received from a pervasive environment, with inputs the same as S_r's but in a different order. Figure 5.7 shows how applying the hash function on the combination of inputs and the output (without sorting them first) can generate two different hash values.

The entry corresponding to a hash key is a link to a list of workflows containing the service used to generate the hash key. Each workflow, in turn, is implemented as another hash table. The key for the workflow hash table is created by combining the input and output URIs of each component service appearing in the workflow along with the "input" and "output" prefix respectively (See Figure 5.8 on page 112). The entry corresponding to

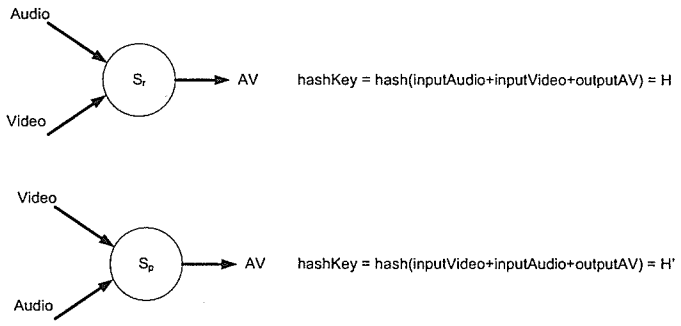


Figure 5.7: Applying a hash function on two services with the same input types but in a different orders

a hash key has multiple fields including a flag indicating whether or not an instance of the service used to generate the hash key has been discovered in a pervasive environment.

The pseudocode for adding a workflow to the repository is shown in Algorithm 1. When a new workflow is added to the repository, the hash value for the combination of the input and output URIs of each component service in the workflow is calculated (line 6). The hash value is searched for in the repository and if the service already exists in the repository, the link to the new workflow is added to the corresponding entry (line 9). Otherwise, a new entry in the repository is created and the hash key as well as the link for the new workflow are added to the newly created hash entry (line 12). This process is repeated for all services in a workflow.

The details of searching for a service in the repository and marking the corresponding workflows are shown in Algorithm 2. When a service, S, arrives from a given pervasive environment, a hash value for the combination of its input and output URIs is calculated and searched for in the repository (lines 4 and 5). If such a service exists in the repository (either by having an exact match or partial match), we can mark it in the corresponding workflow(s) as resolved (line 7). If there is a fully resolved workflow, its *deployed* flag is

Algorithm 1 adding WF to the repository

```
1:  services = get component services of workflow WF
2:  for each service S in services do
3:     inputs = get inputs of S
4:     outputs = get outputs of S
5:     sort inputs and outputs
6:     hashKey = hash(inputs+outputs)        /* "+" means concatenation */
7:     retVal = search(hashKey)
8:     if retVal != null then
9:        add WF to already existing service's entry
10:    else
11:       /* service is not in the repository */
12:       insert(hashKey,WF) into repository
13:    end if
14: end for
```

set and sent to the GD for deployment. The workflow is returned to its original state by unmarking its services (lines 8 to 11).

Example: Assume we have three services as follows:

1. `GetMap`: accepts two inputs of type Address and returns a map between the two addresses in JPG format

2. `ToPS`: accepts an input of type JPG and returns one of type Postscript

3. `Print`: accepts an input of type Postscript and returns no output (but prints the result on a printer)

Having these three services, a workflow, called PrintMapWF, corresponding to a composite service: `GetMap`→`ToPS`→`Print` can be created. Before adding this workflow to the repository, the new workflow must be created as a hash table as shown in Figure 5.8. Figure 5.9 shows the state of the repository after adding only the created workflow[2]. Recall that the services in the stored workflow are abstract place holders for the service instances.

[2]Additional entries would also be present for services from other workflows.

Algorithm 2 search S in the repository

```
1: inputs = get inputs of S
2: outputs = get outputs of S
3: sort inputs and outputs
4: hashkey = hash(inputs+outputs)
5: WFs = workflows corresponding to a given hashkey
6: for each WF in WFs do
7:    mark S in WF as resolved
8:    if WF is fully resolved then
9:       send WF to the GD
10:      set the deployed flag
11:      unmark all services in WF
12:   else
13:      if deployed flag is set then
14:         send WF to the GD
15:      else
16:         no workflow can be sent
17:      end if
18:   end if
19: end for
```

Service's I/O (key)	Resolved
inputAddress+inputAddress+outputJPG	No
inputJPG+outputPostscript	No
inputPostscript	No

Figure 5.8: The PrintMapWF workflow's partial structure containing three services

Service's I/O (key)	WFs
inputAddress+inputAddress+outputJPG	PrintMapWF
inputJPG+outputPostscript	PrintMapWF
inputPostscript	PrintMapWF

Figure 5.9: Additions to the repository after adding PrintMapWF workflow

5.3.2 The Complexity of Repository-based Matching

To discuss the complexity of repository-based matching, I use the notation shown in Table 5.1. To find an exact match between a received service and the services in the repository, I need one search ($O(1)$) in the repository hash table to find workflows containing

Table 5.1: Notation used in calculating the complexity of repository-based matching

Concepts	Notation
Number of services in the repository	n
Number of workflows in the repository	m
Average length of a workflow	l
Average number of inputs	n_I
Average number of outputs	n_O
Average number of subclasses	n_{sub}
Number of super-classes	1

the received service. (This search may require multiple comparisons if there are hash collisions.) Once a service in the repository is matched with the received service, all the workflows must be examined to mark this service as resolved. The average number of workflows that a service will be involved in is $\frac{l*m}{n}$. (This average is just an order of magnitude estimate.) Within a workflow, only one search is needed to find the service in question. Thus, the total number of comparisons required to mark a service in workflows is $\frac{l*m}{n}$. If we assume a static repository without dynamic additions of new workflows, this value is a constant.

If there is no exact match in the repository, I use partial matching which searches for all possible combinations of subclasses of inputs (we have $(n_{sub}+1)^{n_I}$ combinations including inputs and their subclasses) and super-classes of outputs (we assume only one superclass for each type so we have 2^{n_O} combinations including the output and its superclass). In this case, the total number of combinations that must be checked will be $(n_{sub}+1)^{n_I} * 2^{n_O}$. For each combination, a search within the repository is required which can be, in turn, either exact or partial. Since any realistic ontology is not very deep (e.g. see Appendix B), this process will finish quickly.

It will be shown in the next chapter that searching for an exact match in the repository scales well when the number of stored services increases. However, enabling partial matching in the repository search will increase the matching time as the above analysis suggests.

5.4 Input/Output Matching

A repository may not record all the possible compositions that can be made with the announced services (e.g. from a home). For example, when new devices offering previously unseen services arrive on the market, there is no way to have pre-existing workflows involving those services in the repository. As a complementary technique, we therefore also use semantic-based, Input/Output (I/O) matching to find possible compositions not defined in the repository. After ranking sequences found using this technique, a highly ranked subset of them are added, as workflows, to the repository to speed up similar matches in the future.

The game console example described earlier offers many services that can be composed with other available services in the home. It is not possible to foresee all these possibilities and put them in the shared repository. For example, a "store audio video" service offered by the game console might take video and audio signals (SDVSignal and AudioSignal types in the designed type ontology) as its inputs and stores them as an MPG file (MPG type in the type ontology). On the other hand, the standard definition TV cable box might offer the "decode signal" service which takes encoded signal (SDEncSignal type in the type ontology) as its input and produces decoded video and audio signals (SDVSignal and AudioSignal types in the type ontology). These two services are input/output compatible and could be used to generate a composite service "decode signal+store audio video". This service could then be used, for example, to record favorite TV shows. Similar composite

services could be created to store "security" videos coming from a home monitoring camera. Centrally, for the first person installing such a game console in his home there will be no pre-existing workflows in the repository. Thus, I/O matching is required.

Another composite service "play video+upconvert video+display video" could be generated by involving an "upconverter video" service, offered by the game console, which takes a video signal (SDVSignal type) as its input and generates an upconverted video signal (EnhancedVSignal type) as its output. The generated composite service would play normal video files and upconverts them to a better quality to display on a high definition TV in the home. This composition will not exist until a game console and appropriate HD TV are co-located in the same pervasive environment. Thus, again, I/O matching is required.

I/O-based composition, in its simplest form, consists of finding a sequence of services in which the output of a service is type compatible with, and therefore can be sent to, the input of the next service. This simple form of I/O matching is conceptually straightforward but there are some issues that must be addressed:

- The number of possible compositions increases with the number of available services in an environment (i.e. with increasing *service density*). Creating too many services of low interest to the user is clearly undesirable. Ranking the generated composite services in some way will assign them a quantitative value that can be used to discard low-ranked (i.e. non-interesting) composite services and hence reduce number of generated services.

- Doing semantic matching to find input/output compatible services is costly. Calling an inference engine for each matching to find subclass/superclass relationships between inputs and outputs will further increase the matching time. Service enablers

can provide much more computing power to do input/output matching than embedded devices.

- Sometimes the output of a service cannot be sent directly to the input of another one (e.g. when two types are siblings in the type ontology). In this case we must use *glue* services (similar to short-lived services in Ninja [40]) to solve this problem. Short-lived services can be downloaded to and exist on the GD without being bound to any particular device in the environment. Short-lived services can also be used to improve the expressiveness of a workflow. For example, a service can copy the output of one service to multiple inputs enabling concurrent invocation of component services in a workflow.

- A user may be required to provide input for some services and we wish to be able to use such services in compositions. To formalize this process, we define a set of em user-input services with no inputs and one output. Using one of these services to provide user input for a service implies getting some data from a user (e.g. prompting for a selected TV program to record). Similarly, we define *user-output* services with one input and no output that are used in cases where the result of a service are to be presented to the end user in some way (e.g. printing an image).

- A service can also have multiple inputs and/or outputs. This implies that our workflows are not necessarily linear. I discuss such multi-input services in Section 5.4.3.

My approach to I/O matching addresses all these issues.

5.4.1 I/O Matching Using Lookup Tables

To increase the speed of I/O matching at the SE side, my system caches information about different types, prior to the actual matching, in lookup tables. To permit this, I assume

that the ontology remains static during active composition [3]. The ontology is read in advance and the information (i.e. URI, relationships, etc.) about different types are extracted from and stored into lookup tables. In my prototype I use two lookup tables; one for the input parameters of services (*input cache*) and the other for the output parameters (*output cache*). We will eventually see how the information about subclass/superclass relationships between two types are kept in the lookup tables but for simplicity, we first concentrate on exact matching. Since we assume a unique URI for each type that is read from the ontology, we can implement the lookup table as a hash table whose key is the URI of a type.

When a service, S_i, is discovered and sent to the SE, because I assume a common ontology between the pervasive environments and the SEs, its input and output types can be located in the input and output cache respectively and the service, S_i is inserted in the location corresponding to the hash key created for its input and output types. (i.e. a "pointer" to the service S_i is stored in two locations; one pointer from the input cache and the other is from the output cache.) The entries in the input and output cache are header nodes to linked lists connecting all the services that have the same input or output, respectively, depending on which cache we are referring to. This organization allows us to quickly identify all services that may participate in a possible composition based on the types they support.

The implemented input and output caches also maintain information about the subclasses and super-classes of each input and output type. This information is generated using an inference engine, Pellet [76] in our case, operating on the domain ontology shared between services. Having input and output caches in place (i.e. preloaded with ontological information), requests for type matching can be answered without invocation of the inference engine at run-time which is the major source of overhead in I/O-based matching. In

[3]Changes to the ontology require temporary suspension of composition and rebuilding of the lookup tables. This rebuilding can be done in an incremental fashion to minimize suspension time but discussing how this would be done is outside the scope of this thesis.

practice, these tables could be populated in a lazy fashion by making inference engine calls only when necessary.

When a service S_i is received by the SE which is not already in the repository, it is added to the input and output caches. One can then search for services (composite or component) that are composed of acyclic sequences involving the new service. To avoid producing an unduly large number of compositions, I both ensure that the composite service is cycle free and limit the length of compositions. To be able to detect a cycle in result of a composition, I check the composite service to make sure that each service appears only once. I use a hash-based structure to keep track of which services have appeared in what composite services. This structure also minimizes the overhead of cycle checking. Assuming the arrival of service S_i, such composite services can be created using the following steps:

1. *Prefix matching:* Algorithm 3 shows how prefix matching finds I/O compatible sequences. It first looks for the list of services whose output matches, exactly or partially, the input of S_i (line 5). For each service, S_k, in the list, S_kS_i will be a compatible sequence. The sequence S_kS_i is then added to the cache in the same way that S_i was (lines 6 to 9).

2. *Postfix matching:* Postfix matching is presented in Algorithm 4. It finds the list of services whose input matches, again exactly or partially, the output of S_i (line 5). For each service, S_j, in the list, S_iS_j will be a compatible sequence. The sequence S_iS_j is added to the cache as well (lines 6 to 9).

3. If sequences like S_kS_i and S_iS_j are found, a new sequences can be created by concatenating these sub-sequences. (Cycle checking is, of course, performed to make sure that the result of any merged services is cycle free.)

Assume there are three services: S_1 with input type t_1 and output type t_3, S_2 with input type t_1 and output type t_5 and S_3 with input type t_3 and output type t_2. Figure 5.10 shows

Algorithm 3 prefix I/O matching of services with one input/output

1: $input_i$ = get input of S_i
2: $output_i$ = get output of S_i
3: $hashin_i$ = **hash**($input_i$)
4: $hashout_i$ = **hash**($output_i$)
5: $services$ = **search**(outputCache,$hashin_i$)
6: **for each** S_k in $services$ **do**
7: $input_k$ = get input of S_k
8: $hashin_k$ = **hash**($input_k$)
9: **add**($hashin_k$,S_kS_i) to inputCache
10: **add**($hashout_i$,S_kS_i) to outputCache
11: **end for**

Algorithm 4 postfix I/O matching of services with one input/output

1: $input_i$ = get input of S_i
2: $output_i$ = get output of S_i
3: $hashin_i$ = **hash**($input_i$)
4: $hashout_i$ = **hash**($output_i$)
5: $services$ = **search**(inputCache,$hashout_i$)
6: **for each** S_j in $services$ **do**
7: $output_j$ = get output of S_j
8: $hashout_j$ = **hash**($output_k$)
9: **add**($hashin_i$,S_iS_j) to inputCache
10: **add**($hashout_j$,S_iS_j) to outputCache
11: **end for**

the input and output caches corresponding to the inputs and outputs of these three services: S_1, S_2, and S_3. Now, assume the discovery of a new service S_i whose input type is t_2 and output type is t_1. Following the aforementioned steps to find compatible services, we would discover the sequences: S_3S_i, S_iS_1, and S_iS_2. These discovered sequences can be further composed to discover $S_3S_iS_1$ and $S_3S_iS_2$. Figure 5.11 shows the state of the lookup tables after adding the discovered sequences. (Recall that we place a limit on the length of sequences to control the composition length and subsequently the running time of I/O-based composition.) All the cache structures have been implemented using hash classes that handle the collisions that may occur when an entry is added to the cache.

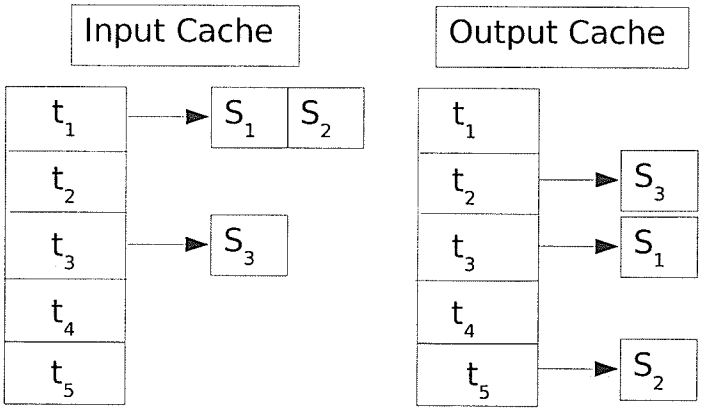


Figure 5.10: Input and output caches corresponding to input and output parameter types

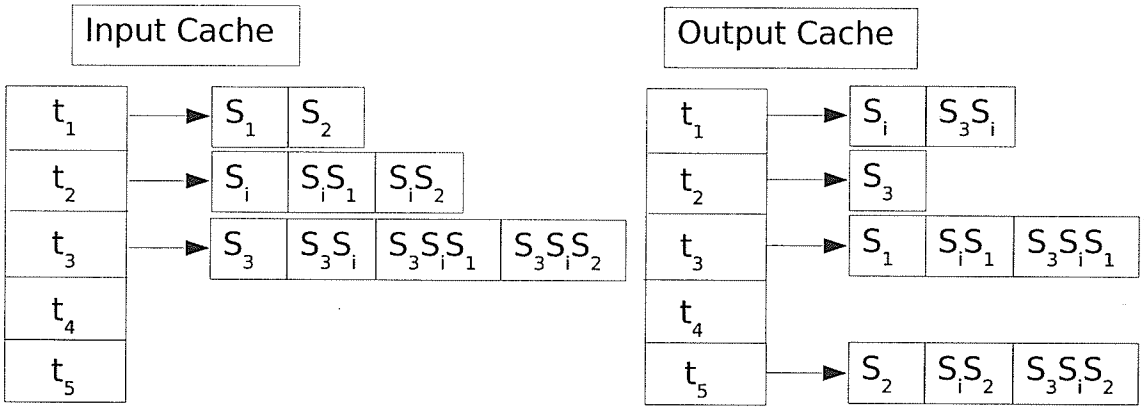


Figure 5.11: Input and output caches after inserting S_i and finding sequences

5.4.2 Complexity of I/O-based Matching

In this section, I determine the average number of possible compositions for a given number of services. Based on these results, I can estimate the overhead of finding all composite

sequences using I/O matching upon the discovery of a new service in a given pervasive environment.

Assume we have N atomic services in a pervasive environment each with one input and one output whose types are taken uniformly at random from an ontology having T data types. Assume further that the assignment of types to inputs and outputs is independent. Also assume that the length of composition is limited to 5. (A value chosen to reflect an expected limit on the length of practically useful compositions.) For simplicity, I assume an ontology without any sub-class/super-class relations between types. (That is, there is no partial matching and hence this gives only a lower bound on the average number of compositions.) Using this information, I can find the average [4] number of sequences of length 2, 3, 4, and 5 (denoted as N_2, N_3, etc. in the following equations):

$$\begin{aligned} N_2 &= \tfrac{N(N-1)}{T} \\ N_3 &= \tfrac{(N-2)N_2}{T} \\ N_4 &= \tfrac{(N-3)N_3}{T} \\ N_5 &= \tfrac{(N-4)N_4}{T} \end{aligned} \tag{5.1}$$

or

$$N_j = T^{1-j} \prod_{k=0}^{j-1} (N-k)$$

To find the number of compositions of length 2, each service is compared with all the other $N-1$ services, thus there are $N(N-1)$ comparisons. Since we have T different data types in the ontology and assuming that the services' inputs and outputs are taken uniformly from this ontology, on average $1/T$ services will be I/O compatible and we should divide $N(N-1)$ by T. If we select $T = 1$, in the worst case, all the services will be I/O compatible and we will have $N(N-1)$ compositions.

[4] By average, I mean an average over samples where each sample is one realization of assignment of inputs and outputs to all N atomic services.

Table 5.2: Comparing the average number of compositions of length 2, 3, 4, and 5 obtained by an experiment to that obtained using formula 5.1

Type of measurement	Avg. number of compositions	95% confidence interval
Experiment	923.6	(878.176,969.03)
Formula 5.1	912.97	-

Similarly, we can calculate compositions of length 3, 4, and 5 in a recursive manner. These equations are just based on N and T so by knowing the number of discovered services and the number of types in the ontology one can calculate the average number of sequences of length 2, 3, 4, and 5.

To assess the validity of formula 5.1, I created 100 different groups of services each having 100 different services with one input and one output. All the types for inputs and outputs were selected from a synthetic ontology with 72 different types. I ran my matching algorithm on different groups of services and collected statistics about the number of generated compositions of length 2, 3, 4, and 5. Table 5.2 compares the results of running the experiment to those obtained from by formula 5.1. The result of the experiment verifies the result of formula 5.1.

Another question that is worthy of answering is how many sequences might be added upon the discovery of a new service. If we assume N services in an environment, and then a new service S_i, is added, the number of newly created sequences of length 2 (i.e. ΔN_2), using the formula in 5.1, will be:

$$\Delta N_2 = \frac{N(N+1)}{T} - \frac{N(N-1)}{T} = \frac{2N}{T}$$

And similarly for sequences of length 3, 4, and 5:

$$\begin{aligned}\Delta N_3 &= \frac{3(N)(N-1)}{T^2} \\ \Delta N_4 &= \frac{4N(N-1)(N-2)}{T^3} \\ \Delta N_5 &= \frac{5N(N-1)(N-2)(N-3)}{T^4}\end{aligned}$$

Using these equations we can find the total number of services after adding S_i:

$$N_i = N + 1 + \sum_{j=2}^{5} \Delta N_j$$

or

$$N_i = \sum_{k=1}^{5} k! \binom{N+1}{k} T^{1-k}$$

Recall that we have $N+1$ services after adding S_i and we have compositions of length 2, 3, 4, and 5. Figure 5.12 shows a graph of the total number of compositions for different numbers of services and available types in the ontology. The figure shows that the number of available compositions grows exponentially with the number of available services. Also we can see from this figure that the size of the ontology (i.e. the number of different types in the ontology) has a great impact on the average number of compositions. This makes sense since by increasing the number of specific data types in the ontology the likelihood of having two services with the same (or similar) data types is lower which in turn decreases the number of compositions.

If we consider, as an example, a typical home with a variety of home appliances, entertainment and computing devices, the number of services, N, might be on the order of a hundred. Each device would normally provide more than one functionality or service. For

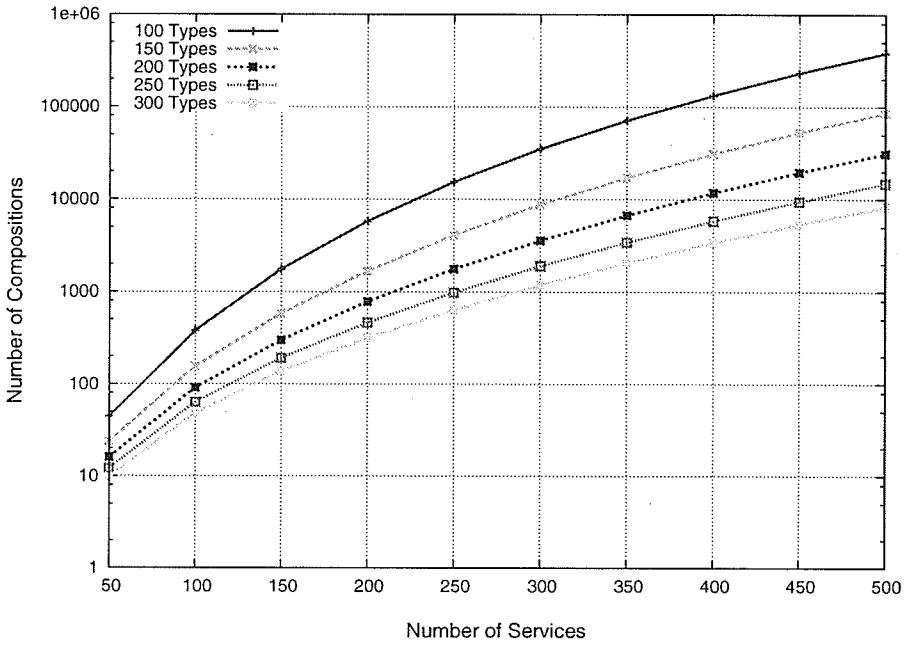


Figure 5.12: The number of possible compositions for different numbers of available services and types

example a cell phone can also play audio files, keep schedules, take pictures, and in some cases run Java programs. The number of types, T, depends on the ontology used in an environment. The real domain ontology developed in this thesis (discussed in Chapter 4) has a very detailed distinction between data types so the number of types, in a complete ontology, would be on the order of a few hundred. This estimate is based on my experience with implemented parts of the ontology and realizing that other real world environments will have different characteristics (in terms of types used in services' inputs and outputs, shape of ontology, etc.).

To compute the average number of comparisons required to find composite services upon the discovery of a new service, I assume N services and T different types. If I further assume that the input and output cache use a balanced hashing scheme, we will have, on average, N/T services in each hash entry. When a service S_i arrives, we have to do prefix and postfix matching (as described earlier). Thus, we need to deal with at most N/T services. Since we already know that these services are type compatible with S_i, we do not need to do any comparison at this point. The same argument holds for postfix matching. However, when we want to merge two composite services, discovered after prefix and postfix matching, such as S_xS_i and S_iS_y (S_x and S_y can themselves be composite services), we must do cycle checking to make sure that our final composite service is cycle free. To detect a cycle in a composite service, as mentioned earlier, I check the composite service to make sure that each component service appears only once. Since I use a hash-based structure to keep track of component services of a composite service, cycle checking does not incur significant overhead.

5.4.3 Services with Multiple Inputs and/or Outputs

So far we have considered only services with at most one input and/or one output. There are some services (e.g. the `GetMap` service discussed in an earlier example) that can accept more than one input and/or generate more than one output. In the lookup table-based approach each such service will be added to different locations (corresponding to each input/output) of either the input and/or output cache.

Simply adding a service to different locations of a cache would solve the problem if the service could generate its output when *either* of its inputs were provided (i.e. supporting an OR presence relation among the inputs). The problem is more challenging if, as typical, a service requires all its inputs to operate (i.e. having an AND presence relation among

the inputs). In the latter case, each service with multiple inputs must know whether or not all its inputs can be provided by some other services. If so, the service can participate in a composition otherwise it must continue "waiting" for services that can provide any, as yet, unprovided inputs. I assume an OR relationship between the *outputs* of a multi-output service. Thus, a multi-output service can participate in a composition using only a subset of its outputs. Such composition, however, will be penalized for the missing output when the ranking of the composite service is calculated (discussed later).

Assume we have a service, S_1, with two required inputs of type t_1 and t_2, respectively. When another service, S_2 with output type t_1 arrives, S_2S_1 cannot form a sequence since S_1 also needs its other input to be provided. Another service, say S_3, with output type t_2 is required to complete the composition. On the other hand, if I assume a service S_1 with one input of type t_1 and service S_2 with two outputs of type t_1 and t_2 respectively, S_2S_1 will be a possible composite service, although one of S_2's outputs has not been used in this composition.

In my implementation, services in the input cache with multiple inputs, maintain links to keep track of their all inputs. In other words, having access to one of the inputs of a multi-input service, provides the necessary links to the other inputs. A similar arrangement is made in the output cache for services with more than one output. Whenever a multiple-input service is received, the code running at the SE side tries to find all the needed services with outputs compatible with the inputs of the received service. If there are not available services for *all* the inputs, those inputs that services have been found for are marked. Later, when other services arrive, if their outputs match with the remaining inputs of a multi-input service, that service can become involved in subsequent compositions with all the necessary services providing inputs as its predecessors.

5.4.4 Ranking Composite Services

The number of composite services created using I/O matching can be overwhelming. This is one of the main challenges of doing fully automated service composition. To select a subset of the generated composite services and present them to the end-user (without overloading the user with a multitude of unuseful composite services) some sort of ranking is needed. For example, the "store file" service offered by our hypothetical game console can participate in a composite service such as "capture image+store file+print file". This composite service may not be of end-user interest if the user already has a service capable of printing a picture directly from a digital camera, i.e. "capture image+print image".

Some work on ranking composite services has been done and I build on this work. Arpinar et al. [18] use *similarity value* (the match type (exact, partial, etc.) between the output and input of two successive services in a composition) as well as *quality rate* (i.e. QoS metrics such as reliability of a service, cost of accessing a service, etc.). They model a composite service as a graph with nodes representing component services and links between pairs of I/O compatible services. They then assign different weights to different edges based on the quality rate and similarity value. Their approach then finds the shortest path in the composite service graph and this results in selecting the "best" service instances for the final composite service. Unfortunately, they do not provide any justification for the selected weights. Arpinar's ranking method mixes parameters that belong to abstract services (i.e. matching quality) with those related to concrete services (i.e. QoS metrics). Furthermore, their goal was not reducing the number of generated composite services.

Chafle et al. [23] consider a technique to select K (an arbitrarily chosen number) workflows from the pool of generated abstract workflows. They use the Hamming distance between two generated abstract workflows. The Hamming distance specifies how many service types in two abstract workflows must be different. Selecting a higher Hamming

distance decreases the number of possible abstract workflows that can be generated. However, they do not consider any similarity value in their decision. Thus, their technique only reduces the number of generated workflows without considering how useful a composite service can be in a given pervasive environment.

The main focus of Chafle et al. is to satisfy QoS requirements. They use three QoS metrics (cost, response time, and availability) associated with each service to find better service instances to use to create a workflow instance (i.e. an abstract workflow with discovered service instances). They calculate the aggregate value of each QoS metric for a workflow and these aggregated values are added up to determine the final QoS value of a workflow. They evaluated their method in terms of its responsiveness to change of QoS parameters.

I use a ranking method which is a combination and extension of these approaches to reduce the number of possible sequences (i.e. abstract workflows) that will be sent to the GDs as well as added to the repository at the SE side. The method is also used to select the best possible service instances when deploying a workflow in a particular pervasive environment. The ranking that I use are:

- Abstract ranking: this method is used to rank different abstract workflows (i.e. composite service templates).
- Concrete ranking: this method is used to find the best possible service instances to use to deploy an abstract workflow.

5.4.4.1 Abstract Ranking

To be able to calculate the abstract rank of a composite service, I use the following values:

1. Matching value (i.e. the type of match between input and output types of two services, either "exact" or "subsume"),

2. The rank of each *component* service (recall that each component service can be either atomic or composite),

3. The length of a composition, and

4. The number of component services with missing outputs.

I refer to the first two parameters collectively as matching quality later in this section.

Similar to Paolucci et al. [63] I assume that an exact match as well as a direct subsumption match have the highest matching value, i.e. 1.0. Indirect subsumption match is assigned a value less than 1.0. I also assume that the rank of a component service is 1.0 if it is an atomic service otherwise its rank is calculated using Formula 5.2.

The Matching Quality (MQ_{ij}) between two component services in a composite service might logically be calculated in either of the following ways:

1. By taking the *minimum* of the matching values between two services and the rank of each component service. That is:

$$MQ_{ij} = min(\text{matching value,rank of each component service})$$

2. By *multiplying* the matching value between two services and the rank of each component service. That is:

$$MQ_{ij} = \text{matching value} \times \text{rank of each component service}$$

Assume two component services S_1 and S_2, with rank 1.0 and 0.8 respectively, participate in a composition S_1S_2 (S_2 is a *composite* service whose rank has already been calculated). Also assume that the matching value between these two services is 0.8 (i.e.

there is an indirect subsumption match between these two services). The matching quality, MQ_{12}, of this composition is either $min(1.0, 0.8, 0.8) = 0.8$ or $1.0 \times 0.8 \times 0.8 = 0.64$ (depending on which of the potential methods is used). Clearly, the latter approach more aggressively penalizes multiple "imperfections" in composite services.

The result of the calculation of MQ_{ij}, will always be a number between 0 and 1. If a composite service has only two component services, MQ_{ij} specifies the total (over the entire composition) matching quality for that composite service. Otherwise, the total matching quality of a composite service, MQ, will be either:

$$MQ = min(MQ_{12}, \ldots, MQ_{(n-1)n}), \quad \text{n=\# of component services}$$

or:

$$MQ = \prod_{2}^{n} MQ_{(i-1)i}, \quad \text{n=\# of component services}$$

depending on the method chosen. If a composite service has branches (i.e. it has a service with multiple inputs and/or outputs), the *minimum* calculated matching quality for branches will be selected to conservatively compute the total abstract rank.

The length of a composition (l) can also be used as a measure to rank composite services. Longer composite services involve many component services and are more prone to failure. Also, a longer composite service has longer running time since each of its component services need to be invoked separately[5]. Finally, long composite services are more likely to have equivalent shorten ones that would be preferred by users. Since the minimum length of a composition is two (involving only two component services), two over the composition length ($L = \frac{2}{l}$) will be a number less than or equal to one and can be easily factored into the calculation of the total abstract rank. If a composite service has branches,

[5]While there can be concurrency among component services, generally they run sequentially. For these reasons, equivalent shorter compositions should be preferred.

the longest branch will be conservatively selected as the composition length.

Composite services involving component services with more than one output and using a subset of outputs, tend to have less appeal for the end-user. For example a "play video" service might have "video signal" and "audio signal". Not using either of these outputs in a composition, in most cases, will result in a less useful (or even non-useful) service. To consider this issue in calculating the total rank of a composite service, I use a factor $M = p^n$ where $p < 1.0$ and n is the number of component services with one or more unused outputs in the composed service.

Having the total matching quality (MQ), length of a composition (L), and number of services with missing outputs (M), the total abstract rank of a composite service can be calculated as:

$$rank_a = \alpha \times MQ + \beta \times L + \gamma \times M, \quad \text{where } \alpha + \beta + \gamma = 1 \tag{5.2}$$

Based on the perceived importance of each factor (i.e. matching quality, composition length, and number of services with missing outputs), different weights can be assigned in calculating $rank_a$. This assignment of weights might be specific to the domain (and corresponding ontology) used.

Once the total abstract rank is calculated, possible compositions obtained using I/O matching are sorted based on the calculated total abstract rank. To be able to send to the GD and add to the repository only a "manageable" number of workflows, I use two threshold values (t_1 and t_2 where $t_1 < t_2$) to prune unnecessary workflows off the list. The threshold values can be built into the system or entered by an SE expert. Again, selection of those thresholds may be domain specific. Composite services whose rank is greater than t_2 will always be sent to the gateway device and added to the repository. Composite services whose rank is between t_1 and t_2 should be "assessed" at the SE side before deciding whether to

keep or prune them. Finally, composite services with ranks below t_1 will be discarded. The next chapter discusses how threshold values can be computed for different scenarios.

Usage-based Abstract Ranking

The frequency of use of a composite service in a pervasive environment can also be used to reduce the number of workflows maintained in the repository and hence, ultimately presented to users. The SE which is responsible for finding composite services, can also keep track of the use of deployed composite services. Whenever a composite service is accessed within a pervasive environment, the GD can send the relevant information about this access to the SE. Each GD is associated with one of a number of pre-selected user categories (e.g. "expert", "average", and "novice"). The number of categories is flexible and, depending on the deployment environment, different number of categories might be selected. A GD will be assigned to one of these categories based on the level of technical sophistication of its managed environment's occupants. [6] In this way, "similar" pervasive environments, based on predefined preference settings and/or the usage information about deployed composite services, can be identified and corresponding GDs (in the same category) can be grouped together.

Initially, the SE composes services and ranks them. Only highly-ranked composite services (those whose ranks are above t_1) are sent to the pervasive environment for deployment and added to the repository. However, it is possible that some of these highly-ranked services are actually not of end-user interest or may be of interest for only a very short period of time (e.g. until they are supplanted by more desirable compositions, possibly related to new multi-function devices). It is also possible to have a composite service which might be frequently used in some environments but which has a lower rank (i.e. between t_1 and t_2). The SE will store such lower ranked composite services in the repository to

[6] The category can be selected by the user if such information is not automatically available.

be able to collect their usage statistics for different environments. Then, the SE, before sending a composite service with a low computed rank to an environment, can check to see if other similar environments (i.e. GDs in the same category) have already used the newly-composed service or not. Based on usage information in similar environments, it is therefore possible for a composite service with a low rank to be selected and sent for deployment. Of course, to bootstrap this process some users must "try out" such mid-ranked compositions. Expert users are more likely to do this and a mechanism where by their experiences may be exploited will be described shortly.

Input/Output matching in the game console might result in a long composite service such as: "decode signal+store video+play video+display video". This composite service will get a relatively low rank because of its length. The generated long composite service, however, acts as a Personal Video Recorder (PVR) which records a TV program while it is playing and therefore could be very desirable. The information about the usage of such a long composite service in other homes can be collected at the SE side to infer that this composite service is indeed useful despite its comparatively low rank. The collected information can be used to adjust the calculated rank and this long composite service will therefore later be deployed in other pervasive environments.

To implement this filtering method based on the use of composite services, additional information must be stored for each workflow. The structure of an entry for each workflow will be similar to that shown in Figure 5.13. In particular, for each workflow in the repository, its computed rank, number of deployed instances, and number of times an instance has been accessed must also be stored. The number of deployed and accessed instances of a workflow should be maintained for each GD category to allow differentiation of usefulness for different types of users.

Consider service composition again which starts when the SE receives a description of new services from a GD. If a composite service using the new component service is

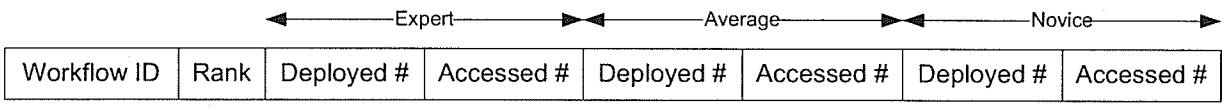


Figure 5.13: Information stored about each workflow per each GD category (e.g. expert, average, and novice)

discovered in the repository and its rank is above the selected threshold value, t_2, it will be sent (without further check) to the GD for deployment. If its rank is between the two threshold values (i.e. it is a "mid-ranked" composite service), the discovered composite service will be sent to the GD under either of the following conditions:

- The requesting GD is in the "expert" category. The users of a GD in the expert category are assumed to be technical enough to be exposed to composite services that are not highly ranked. Since they can easily and quickly select the services they actually want from the larger list presented to them. The usage information about mid-ranked composite services will subsequently be collected from these "technical" users, and can be used to decide if a mid-ranked composite service should be sent to other, less technical, environments or not. Users in the expert category thus provide boot strapping support for the system as a whole.

- The requesting GD is not in the "expert" category but the number of accesses to this composite service in environments managed by a GD belonging to the same or the next more technical category is high enough that the composite service can be sent to the requesting GD. For example if the requesting GD is in the "novice" category and the rank of discovered composite service is not high enough, the service will only be deployed on the requesting GD if either the same service has already been deployed in the "novice" category or the number of accesses to this service is high enough in

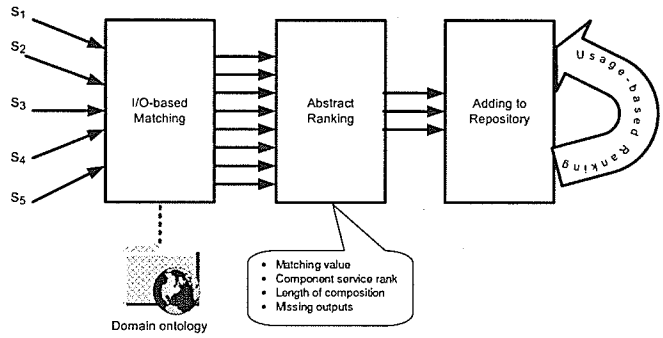


Figure 5.14: The process of composing services and the different stages of filtering them

"average" GDs.

Once the composite service is sent to a GD for deployment, statistics about the deployed workflow are updated accordingly at the SE side.

Later, when a composite service is used in a pervasive environment, the GD sends information about the use of the composite service (e.g. number of times invoked) along with its own identification to the SE. The SE, after receiving the information, updates the related usage information for the appropriate category in the repository.

Figure 5.14 shows the different stages of composing and later filtering the generated composite services. It starts with I/O-based matching which usually results in many compositions. The generated composite services are ranked based on different criteria before adding them to the repository. Once they are deployed in pervasive environments, usage information is collected and the computed ranks are further revised.

5.4.4.2 Concrete Ranking

Concrete ranking is similar to abstract ranking but considers service instances instead of service types. Hence, it involves more QoS related metrics. The following criteria are used to compute the concrete rank of a composite service:

- Service type: a service can be offered by either a mobile or stationary host. An availability metric (see the next item) factors into the service type in calculating the overall concrete rank. Stationary services are preferred over mobile ones due to their enhanced, relative availability.

- QoS metrics: different QoS metrics for composing services have been proposed by Zeng et al. [86]. I use a subset of the metrics proposed which were also used by Arpinar et al. [18] as well as Chafle et al. [23]. These metrics are: availability (A), cost of accessing a service (C), and response time (R). Generally speaking, these metrics can be calculated by considering service type (mobile or stationary), amount of load on the device hosting a service (low, moderate, or high), type of communication link between the host device and the GD (low bandwidth or high bandwidth), and available resources (processing power and memory, etc.) on the host device. Using these measurable parameters, a number can be assigned to each QoS metric. The details of gathering and representing these metrics are not considered in this thesis.

Service type and QoS metrics are computed for each component service. To be able to find the aggregated value of each criterion (including QoS metrics) for a composite service (e.g. to find the availability of a composite service containing several component services with different availability rates), I use the approach of Chafle et al. I multiply values for availability of each component service (these are values between 0 and 1) to calculate the availability (A) of a composite service:

$$\mathcal{A} = \prod_i A_i \quad \text{, where } A_i \text{ is the availability of service } S_i \tag{5.3}$$

The cost ($\mathcal{C}$) and response time ($\mathcal{R}$) of a composite service, on the other hand, are computed as the sum of the costs and response times of each component service, respectively. Since these values can be greater than 1, they must be normalized to be conveniently incorporated in a formula with $\mathcal{A}$. Assume C_{max} and C_{min} are the maximum and minimum cost of accessing a service in an environment. Also, assume R_{max} and R_{min} are the maximum and minimum response time for a service. The normalized values of cost and response time for a composite service can then be simply calculated as:

$$\mathcal{C} = \frac{\sum_i C_{max} - \sum_i C_i}{\sum_i C_{max} - \sum_i C_{min}} \quad \text{, where } C_i \text{ is the cost of accessing service } S_i$$

$$\mathcal{R} = \frac{\sum_i R_{max} - \sum_i R_i}{\sum_i R_{max} - \sum_i R_{min}} \quad \text{, where } R_i \text{ is the response time of service } S_i$$

If a composite service has branches, the minimum of the calculated values of $\mathcal{A}$, $\mathcal{C}$, and $\mathcal{R}$ for each branch are conservatively selected. The overall concrete rank of a composite service based on the selected criteria can then be calculated as:

$$rank_c = W_A \times \mathcal{A} + W_C \times \mathcal{C} + W_R \times \mathcal{R} \tag{5.4}$$

where W_A, W_C, and W_R are different weights assigned based on the importance of each criterion. The value of $rank_c$, as we will see in Section 5.5, can be used as an overall metric to sort different instances of a workflow. If one wants $rank_c$ to be between 0 and 1, the

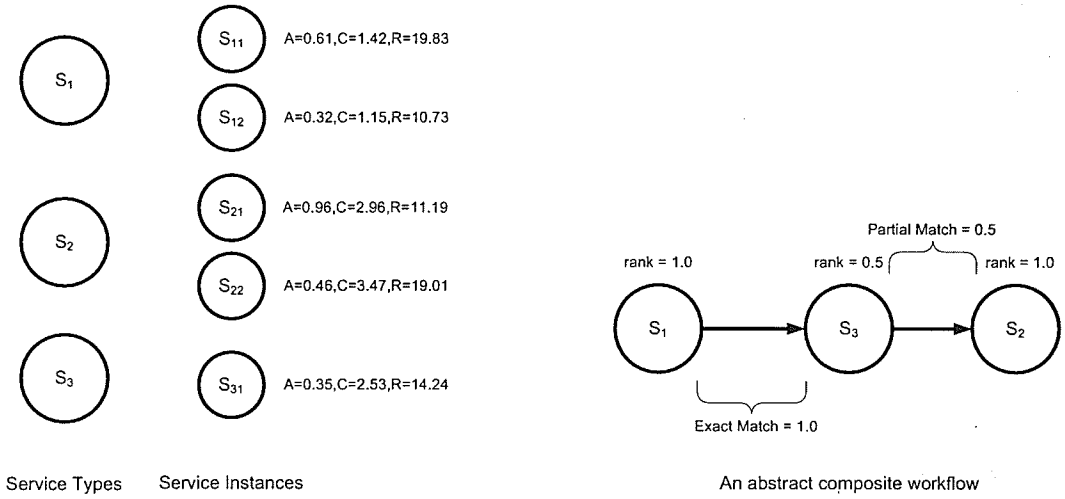


Figure 5.15: Three service types and corresponding service instances with assigned QoS values

aggregate value of weights must be equal to 1. This concrete ranking was not implemented in the current prototype.

Example: In this example, I show how abstract and concrete ranking work. Let us assume there are three service types (i.e. abstract services) S_1, S_2, and S_3 that can create a composite service $S_1S_3S_2$. The matching between S_1 and S_3 is exact and the matching between S_3 and S_2 is partial. The rank of S_1 is 1.0, S_2 is also 1.0, and S_3 is 0.5. Figure 5.15 (right hand side) shows this composite service.

To calculate the abstract rank of this composition using Equation 5.2, the Matching Quality (MQ) of each pair of this composition (i.e. S_1S_3 and S_3S_2) as well as the whole composition must be calculated first. Using the first formula for matching quality, we have:

Table 5.3: The calculated concrete rank for the composite service example shown in Figure 5.15

Composition	$\mathcal{A}$	$\mathcal{C}$	$\mathcal{R}$	$rank_c$
$S_{11}S_{31}S_{21}$	0.20	0.67	0.49	0.45
$S_{11}S_{31}S_{22}$	0.10	0.63	0.23	0.32
$S_{12}S_{31}S_{21}$	0.11	0.70	0.79	0.53
$S_{12}S_{31}S_{22}$	0.05	0.65	0.53	0.41

$$
\begin{aligned}
MQ_{13} &= min(1.0, 1.0, 0.5) = 0.5 \\
MQ_{32} &= min(0.5, 0.5, 1.0) = 0.5 \\
MQ &= min(MQ_{13}, MQ_{32}) = min(0.5, 0.5) = 0.5
\end{aligned}
$$

The length of this composition is three and L in Equation 5.2 will be 0.67. Having weights $\alpha = 0.5, \beta = 0.25$, and $\gamma = 0.25$ and considering the fact that there is no service with more than one output, the abstract rank of this composition is:

$$rank_a = \alpha \times MQ + \beta \times L + \gamma \times M = 0.5 \times 0.5 + 0.25 \times 0.67 + 0.25 \times 1.0 = 0.67$$

The concrete rank of this composition can also be calculated using the service instances shown in the left side of Figure 5.15. In this example, the values for availability (A), cost (C), and response time (R) are selected uniformly from [0.2,1.0], [1,5], and [10,20] respectively. I also assume that service types S_1 and S_2 each have two instances whereas service type S_3 has only one instance. The concrete rank for the given composite service can be calculated using Equation 5.4. I chose weights of 0.33 for each factor to make the final concrete rank a number between 0 and 1. Table 5.3 shows the result of this calculation.

5.5 Composite Service Deployment

To have a fully automated service composing system, discovered composite services (either from the shared repository or by performing I/O matching) must eventually be deployed in a pervasive environment. The GD must deploy the new composite service in such a way that other existing devices/services in that environment can make use of it. A bundle within OSGi (see Figure 3.3), the "OWLS Receiver", is responsible for receiving OWL-S descriptions of composite services from the SE which sends the selected workflows to the GD. The GD keeps a record of received workflows along with their associated concrete ranks. Thus, the GD tracks how many instances of a given abstract workflow have been deployed and what the concrete rank of each deployed workflow is. This information could be used to help the GD to select a better replacement workflow in case of a failure but this feature has not been implemented in the prototype.

Once the description of a composite service has been received by the GD, the OWLS Receiver creates a corresponding wrapper (a virtual service) for each available protocol in the corresponding pervasive environment (e.g. UPnP, Jini, etc.) by calling corresponding bundles within OSGi (See Figure 5.16). The UPnP wrapper creator bundle, for example, creates a UPnP service (using `createUPnPDevice` in the figure) that knows how to invoke the newly received composite service and registers the UPnP version of the service in the OSGi dictionary. Each wrapper will then advertise the composite service using one of the available protocols. These operations are performed by the GD.

The created wrapper services are registered within OSGi and are exported by the appropriate driver bundles outside of the OSGi environment, (i.e. the pervasive environment). Devices and other services available in the pervasive environment will detect the presence of this new service and can then make use of it. When a device/service (outside the OSGi framework) invokes a wrapper service that is available in the environment, the request is

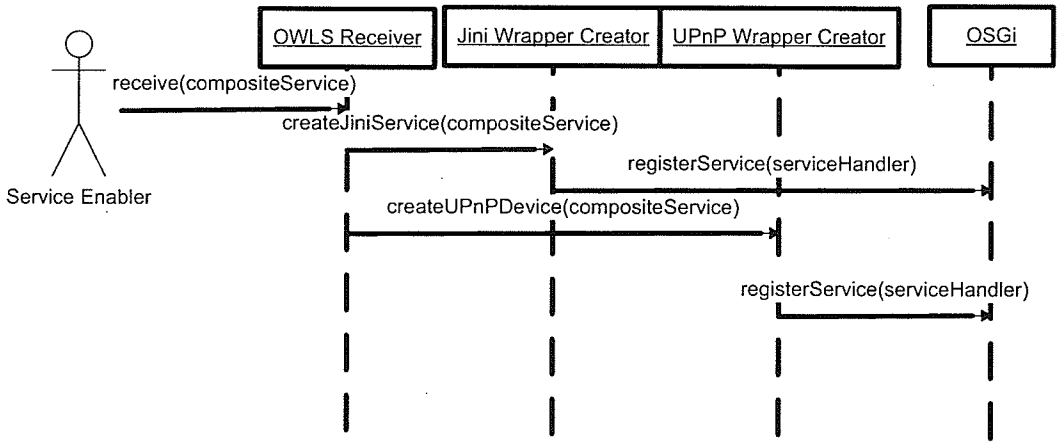


Figure 5.16: The sequence diagram of creating wrappers within the GD

forwarded, by the corresponding driver bundle, into OSGi and another bundle inside OSGi, the OWLS Executer, calls the component services of the composite service. Recall that each component service has its OSGi grounding and the grounding has enough information to contact the actual service outside of the OSGi framework through a driver bundle. If the invoked service has an output, the output will be returned to the calling service via the wrapper. Invocations of the same logical composite service via different wrappers count, collectively, towards the frequency of use totals for the service that are maintained by the GD and sent, periodically, to the SE. Figure 5.17 shows a sequence diagram of calling a composite service by a user within a pervasive environment.

The OWLS Executor bundle must be able to handle exceptions that occur while executing a composite service. As mentioned in Section 3.3, an exception can happen as a result of a fault in a service or temporary or permanent unavailability of the service. The OWLS Executor bundle will either find an alternate composite service, if such a service exists, or return an error to the requester if there is no alternate service. Since the GD keeps

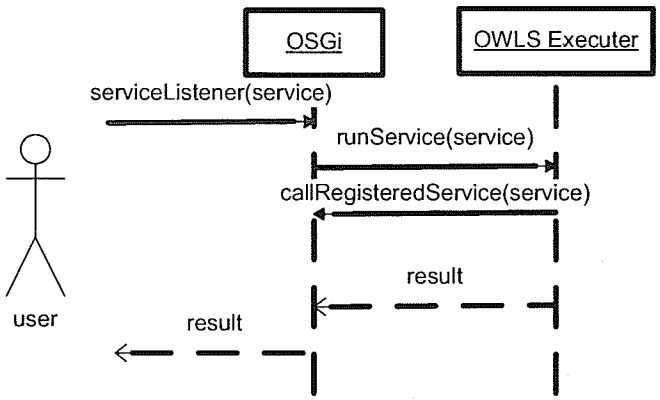


Figure 5.17: The sequence diagram of calling a composite service by a user within a pervasive environment

all the composite services created from the same workflow (i.e. different composite services using different instances of component services), it is possible to find an alternative workflow simply by looking at the abstract description of the failed composite service. The GD also keeps the concrete rank of each composite service which is useful in selecting the best available alternate workflow. Currently the prototype just returns an error in the case of component service failure.

Chapter 6

Experimental Results

To evaluate my system I implemented a small scale proof of concept prototype based on the components shown in Figure 3.3. I Implemented all the GD components using the Oscar [3] implementation of the OSGi framework on a normal PC with a Pentium IV 2.6GHz CPU and 512MB RAM, running RedHat Linux 9.0. The PC acts as both the GD and service enabler in the prototype system. I added OSGi groundings to the OWL-S API [76] so each service defined by OWL-S can be accessed within OSGi. Further, I have tested the system using a variety of emulated UPnP devices (e.g. emulated fridge, TV, and camera running on both normal PCs and PocketPCs) and several sample Jini services (e.g. binary switch, reading status of a device, etc. running on a normal PC) which are all fully functional within the prototype.

The implemented prototype can make effective use of both UPnP and Jini services registered within OSGi as components of composite services. Further, it is extensible to other "middleware systems" (e.g. HAVi) through the addition of required driver bundles. We have experienced no difficulties in discovering and creating composite services using UPnP and Jini services. My prototype has also been used by another student to create composite services out of services offered by X10 [28] devices. X10 devices communicate with my

composition system through Jini and UPnP proxy services. All created composite services are also registered as OSGi services and can therefore be used in subsequent service compositions (i.e. composite services can be built consisting of both existing non-composite and composite services). Additionally, the deployment of composite services as UPnP or Jini services (as well as OSGi services) is implemented. This allows other (non-OSGi) services in a pervasive environment to seamlessly make use of the newly created composite services. The created UPnP wrappers can be detected and later controlled by, for example, a UPnP control point running on a PocketPC. Such a control point for PocketPCs has been also implemented as part of my prototype.

For a fully automated composition system to be considered effective and hence attractive to both end users and SEs, it must be able to correctly discover new compositions in realistic environments, it must be able to do this efficiently, it must not "overwhelm" users with undesirable composite services and it should be useful in different pervasive environments. To assess my system and ensure that it met these goals, I tested the implemented prototype in three different ways:

1. Using real world scenarios and a realistic domain ontology derived from actual devices to make sure that the prototype works as expected. To achieve this goal I created scenarios for adding different devices to a home area network and checked the result of the composition (after ranking) manually. These different scenarios being based on a realistic ontology, to some extent, show that my prototype system is capable of handling real environments.

2. Emulating the real system using synthetic services with an artificial ontology to collect data about average matching time, average number of services, etc. This collected data is used to do scalability testing.

3. Using simulation to show that the proposed architecture is deployable in different

pervasive environments. Their required data to setup the simulation was collected by emulating the system and thus grounds the simulation, basing it on actual performance numbers.

6.1 Real World Scenarios

Real world scenarios use the real domain ontology developed in this thesis (See Chapter 4 and Appendix B). The scenarios are taken from a home area network. Home area network scenarios were chosen because a multitude of devices may exist in a home making it a rich environment for composition.

Scenario 1: In this scenario I assume an empty home and add a TV, a cable box, a DVD player, and an MP3 player in that order. (The details of the services provided by these devices are presented in Appendix C.) After sending service descriptions to the prototype system, 89 different composite services can be generated. (The composed services are shown in Appendix D.)

The result of composition was then manually checked to see what type of compositions were created. Some of composed services were trivial and some of them were not. Composite services, overall, show possible usage of available devices/services in an environment. Then, different parameters of the ranking function (i.e Formula 5.2) were selected in a way that useful composite services get higher ranks with a clear distinction (if possible). Table 6.1 shows values for weights and parameters of the ranking function.

Ranking the generated composite services using the values from Table 6.1 resulted in a set of composite services with relatively clear "cut points". To discard compositions with ranks below a particular threshold, I manually examined the calculated ranks to find appropriate cut points. Some of the calculated ranks and descriptions of the service characteristics used in determining those ranks are shown in Table 6.2.

Table 6.1: Values selected for parameters and weights of Formula 5.2

Weight/Parameter	Value
α	0.50
β	0.25
γ	0.25
MQ	1 (exact), 0.8 (non-direct subsumption)
L	$2/l, l = 2, 3, ...$
M	$0.25^n, n = 0, 1, ...$

Table 6.2: Computed ranks associated with different composition characteristics

Rank	Description
1.0	Composition of length 2 having exact match
0.92	Composition of length 3 having exact match
0.90	Composition of length 2 having non-direct subsumption
0.88	Composition of length 4 having exact match
0.85	Composition of length 5 having exact match
0.82	Composition of length 3 having non-direct subsumption
0.81	Composition of length 2 having a service with a missing output
0.73	Composition of length 3 having a service with a missing output

In Table 6.2 compositions with rank values less than 0.73 are very unlikely to be of end-user interest and can therefore probably safely be discarded. Composite services whose ranks are between 0.73 (inclusive) and 0.92 might be assessed at the SE side to decide whether it is worth sending them to a pervasive environment or not. This would be done based on usage information about these "mid-ranked" services collected from different environments (see Section Usage-based Abstract Ranking on page 132) Finally, composite

services whose ranks are between 0.92 (inclusive) and 1.0 are sent to a pervasive environment for deployment. Of course, composite services whose ranks are greater than or equal to 0.73 are added to the repository. As described earlier, it is possible that some environments will not use one or more highly-ranked composite services. For example, a composite service involving a cell phone for taking pictures or recording a video will not be appreciated by a novice user who uses his/her cell phone mainly for talking purposes. Usage information about these services can also be used to filter the mid-ranked composition and this prevents sending those composite services to environments where they are not used frequently.

The results of the ranking function, and consequently the selected threshold values, are related to general characteristics of a composite service regardless of the type of component services. Specifically, the results are only dependent on the relationships between the input and output types of components in a composite service, their length, and whether or not they have missing outputs. Thus, these threshold values, as shown in Table 6.2, are not limited to a particular domain and can be applied across different domains. The scenario B in Appendix D shows the application of the ranking function and threshold values in a meeting room environment which is different from a home area network.

Applying the ranking function with the selected threshold value of 0.8 filters out 66% of the composite services generated in Scenario 1. Collecting the usage information about deployed services could further reduce the number of deployed composite services.

Scenario 2: In this scenario I assume having a TV, a cable box, a DVD player, and a digital camera instead of an MP3 player. Running my prototype system with the services offered by the assumed devices results in 80 different composite services. The calculated ranks for the generated composite services are similar to the previous scenario and the same threshold values can be used in this case as well. By selecting a threshold value of 0.8, 69% of the composite services are filtered out.

Scenario 3: In this scenario I assume having a TV, a cable box, an MP3 player. I then add a game console (with the capabilities described earlier) to the assumed environment. After sending service descriptions to the prototype system, 97 different composite services could be generated. Applying the same abstract ranking function with a threshold value of 0.8 results in filtering out 73% of generated composite services.

I also did a few other "real-world scenario" experiments and their results are presented in Appendix D. Due to the type and number of services in these scenarios, only a few composite services with a relatively short length have been generated. In more general scenarios (e.g. Scenario 1), I had composite services of maximum length five. This length was selected to avoid very long composition times and can be set to a large number.

I also compared my implemented prototype system with a system developed by Sirin et al. [78]. I loaded my described services (from Scenario 1, for example) to Sirin's system to ensure that I can generate the same set of composite services. Then, I used the provided user interface to manually build a composite service using the loaded services. Since my prototype system uses their API for semantic matching, the set of composite services generated by both systems are, not surprisingly, the same. Sirin's system, however, is completely manual and is not capable of incorporating services with multiple outputs in a composition.

6.2 Emulating the System

Once I obtained confidence that the implemented prototype system was capable of handling real-world scenarios, I decided to test the prototype to collect data about average matching time, average number of services, etc. used to do scalability testing. I performed this test by emulating the real service composition system using a synthetic ontology encoded in OWL containing 72 different types. This number of types is comparable to what I have in the "real" domain ontology. To capture different possible structures of a real domain

ontology, I organized the types in four different ways: 1) as a one-level hierarchy (i.e. all types are direct subclasses of class *Thing* in OWL) shown as "Ontology 1" in the figures in this section, 2) as a two-level hierarchy shown as "Ontology 2" in the figures, 3) as a three-level hierarchy shown as "Ontology 3" in the figures, and 4) as a four-level hierarchy shown as "Ontology 4" in the figures. Each ontology has equal numbers of types in each level. For each ontology, I then created 10 different sets of services each of which having 100 different service descriptions (encoded in OWL-S) with a single input and a single output. (The input and output types were selected uniformly at random from the types in the synthetic ontology.) The URIs for the description of each service were then sent, one at a time, to the program acting as the SE to find possible composite services. The program first used repository-based matching and then, if it could not find any workflows, it used I/O-based matching. The program goes through the entire process of creating a composite service (i.e. doing all the required comparisons) up to creating the description of the composed service (i.e. encoding the generated workflow in OWL-S). Thus the measured time for doing a particular operation is a reflection of real world costs. Whenever a sequence is found using I/O-based matching it is added to the repository for future use. This experiment further shows the capability of my prototype in handling service descriptions specified by different (synthetic) ontologies. The synthetic ontologies gave me the required flexibility to change their properties (e.g. their shapes, number of types, etc.) as desired.

The average required time to find a composite service using I/O-based matching upon discovery of a new service for each domain ontology is shown in Figure 6.1. The results were obtained by taking the average of running 10 different service sets each having 100 services whose inputs and outputs were selected from each ontology. As the figure shows, and as was expected, the required time to find a composite service increases with the number of available services in a pervasive environment. The reason for this is the increasing number of possible combinations of services that can be involved in a composite service.

Figure 6.1 also shows that multi-level ontologies incur more overhead than a flat ontology. This is because partial matching requires checking sub-class/super-class relations between inputs and outputs and this has high overhead.

Figure 6.2 shows the total number of possible compositions of length between 2 and 5 for each test case in Figure 6.1. This figure explains (by showing the number of generated compositions given a number of available services in the system) the amount of time that has been spent for I/O matching as reported in Figure 6.1.

The average time required to find a sequence using repository based matching upon discovery of a new service is presented in Figure 6.3. The difference between the time required to find a composite service using I/O matching versus finding it using the repository is striking (few hundred milliseconds verses 10 milliseconds). Also the number of available services in a pervasive environment does not have an impact on the matching time, which suggests repository based matching will be more scalable. These properties of repository based matching, make it an effective way of supporting larger, more complex pervasive environments containing potentially many services.

A similar experiment was done with a mixture of services having one and two inputs. In this experiment, 1/3 of the services had two inputs and the rest had only one. All the services had a single output. Corresponding to each ontology, I created 10 different sets of services each having 100 services (as described earlier) and sent them to the program acting as the SE to find possible compositions. Figure 6.4 shows the average time required to create sequences of length between 2 and 5 using only I/O matching. Comparing this figure with Figure 6.1 shows a significant increase in composition time. The comparison suggests that I/O matching of multi-input services does not scale well, as the number of services in a pervasive environment increases. Even with relatively few expected multi-input services, this is a potential concern. The number of detected sequences of length 2 to 5 is presented in Figure 6.5.

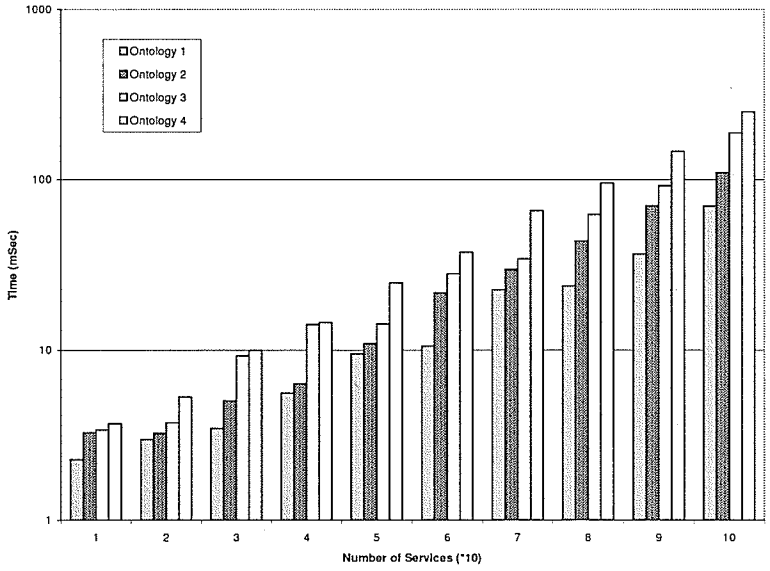


Figure 6.1: Average time required to create a sequence using only I/O matching

The time required to find a composite service using repository-based matching when dealing with multi-input service is largely unchanged as shown in Figure 6.6. Thus, the value of repository-based matching for large pervasive environments is clear. Further, the importance of only having to do I/O-based matching infrequently is also clear.

6.2.1 Applying the Ranking Function

In this section I present the results of applying my abstract ranking method, discussed in Section 5.4.4, to the composite services generated using the synthetic ontologies and component services. To calculate the abstract rank of a composition, as mentioned earlier, I use the rank of each service (calculated using Formula 5.2 if it is not atomic), the type of matching between two component services (exact or subsume), the number of component

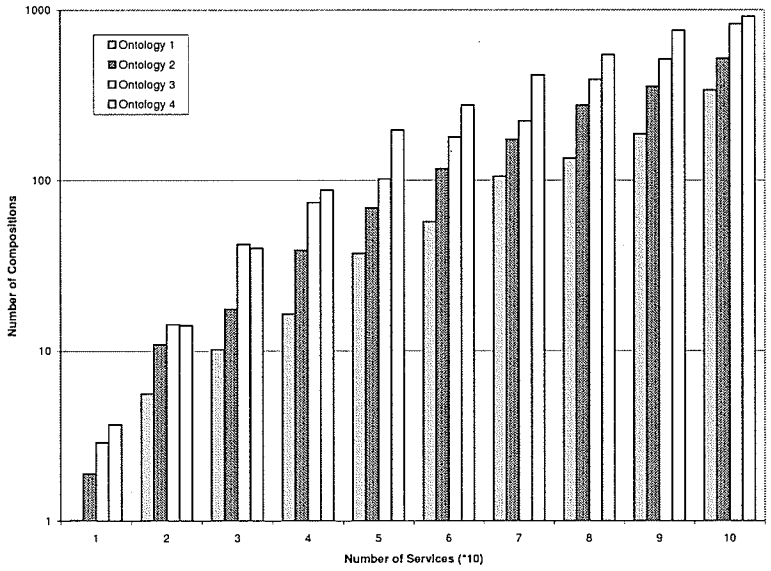


Figure 6.2: Number of possible sequences of maximum length 5

services with missing outputs (i.e. component services having more than one output and only a subset of the outputs have been used in the composition), and the length of the composition. To setup this experiment, I generated 10 different sets of services each having 100 service descriptions for each of the two-level, three-level, and four-level ontologies discussed earlier in this chapter. I created 10 different sets of services for each ontology to make sure that the randomly generated input and output types did not have transient effects on the results.

Table 6.3 shows the number of generated compositions before and after applying the abstract ranking function (Formula 5.2). To select the better compositions I chose a threshold value of 0.8 (based on my experience with the real world scenarios) and all the compositions with a rank below that threshold value were not considered in counting the total compositions. Table 6.3 shows that using the ranking function with a threshold value of

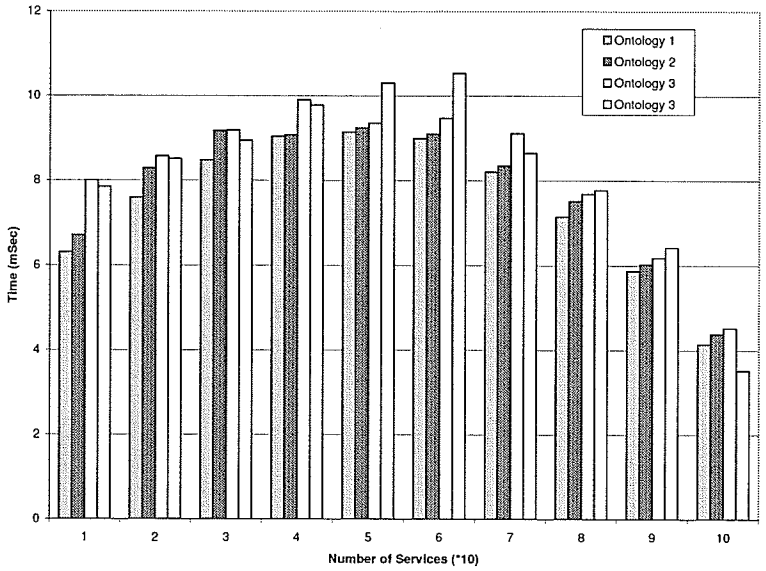


Figure 6.3: Average time required to find a sequence using only the repository

0.8 can remove up to 66% of the generated composite services. This finding is consistent with the results obtained by ranking the generated composite services using the real domain ontology.

Different parameters (i.e. matching type, having component services with missing outputs, and length of composition) are considered in calculating the abstract rank of a composite service. Furthermore, penalty values in the ranking function are fine tuned based on the real-world scenarios. Thus, it can be safely assumed that by selecting a proper threshold value, important composite services will not be removed from the list of services that is sent to the GD for deployment.

By reviewing the numbers in Table 6.3 it is clear that the number of generated compositions increases with an increasing number of levels in an ontology. This is a direct result of having more subclass relations in an ontology that has a deeper structure. However, the

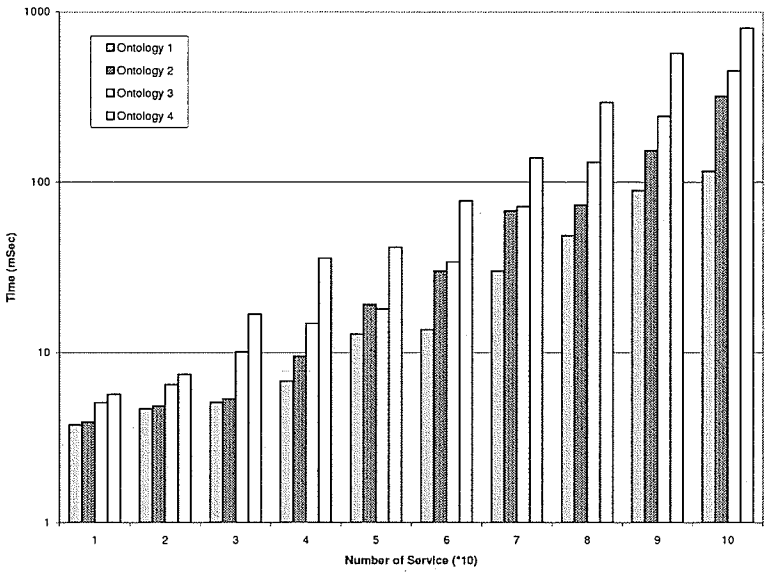


Figure 6.4: Average time required to create a sequence using only I/O matching (1/3 of services have two inputs)

Table 6.3: Results of using different ontologies for composition and applying ranking method and a threshold value

Item	Total Composition	Total With th=0.8
Ontology 2	1577.9	849.4
Ontology 3	2364	999.8
Ontology 4	3244	1078.4

type of matching tends to be more subsume which decreases the matching quality. Thus, the ratio of compositions having a rank higher than the selected threshold decreases. Given that type ontologies tend to be relatively shallow the computation time required in the real-world should be manageable. It would also be feasible to employ a "processor farm" to

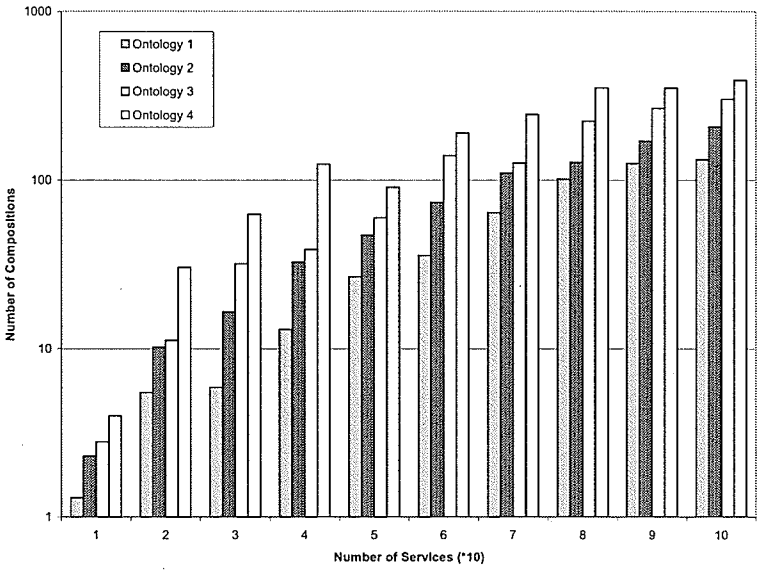


Figure 6.5: Number of possible sequences of maximum length 5 (1/3 of services have two inputs)

help speedup matching if necessary since different sub-trees in the type ontology could be explored independently. Doing matching at the SE makes this feasible. The availability of such resources in pervasive environments, however, is very unlikely.

6.3 Simulating the System

To study the long-term behavior of my composition system and to collect statistics about the different parameters of the system, I simulated the system using the SSJ [5] simulation package. The input parameters of the simulation were service arrival rate (λ) and average time a received service stays in the list of available services ("service lifetime")(μ). Running the simulation system with a large number of service arrivals without removing services from the system results in explosion of the system. Therefore, a lifetime for each

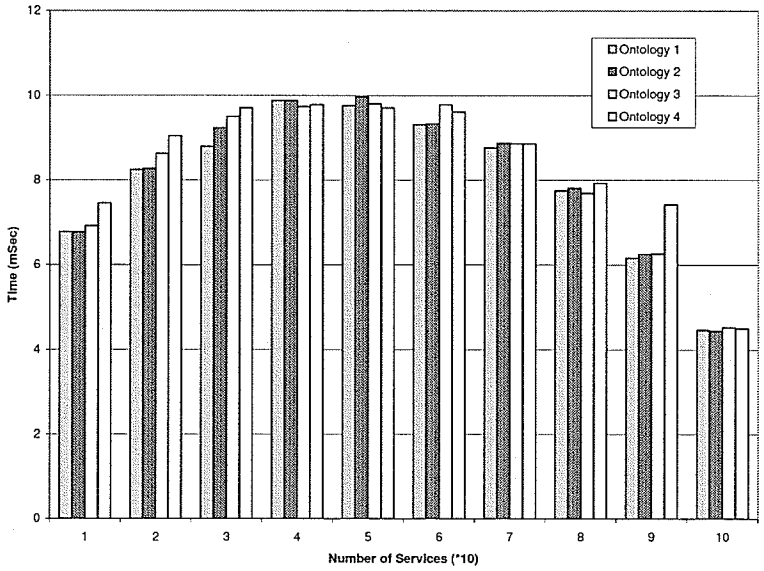


Figure 6.6: Average time required to find a sequence using only the repository (1/3 of services have two inputs)

service must be assigned to balance the number of active services in the system. In the real-world implementation, only services coming from the same GD will be considered for composition and assigning a lifetime might not be necessary.

To measure the performance of the system, I collected statistics on the average number of services waiting for I/O matching, the average number of services received from pervasive environments, and the average time required for I/O matching. The required matching times for repository-based matching, as shown earlier, are almost independent from the number of available services. Thus, I study only I/O-based matching using a simulation model.

To avoid very long matching times, I assumed a maximum of 150 available services. (Testing my system with more than 150 services could sometimes result in a very long

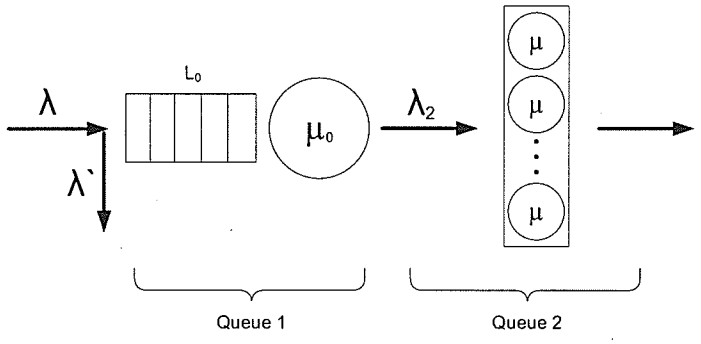


Figure 6.7: Simulation model of the service composition process at the SE side. μ_0 is the composition time and μ is the service residence time.

matching time. Figure 5.12 on page 124, somehow, shows this fact as well.) I only consider I/O matching (no repository matching) in my simulation and therefore my results show the worst case. Figure 6.7 shows the simulation model of the service composition process taking place at the SE side. This model is an approximation of the real system and captures the important aspects of it. The first queue, Queue 1 in this figure, is a $M/G/1/K$ queue [16] in which the buffer size (and consequently the arrival rate to the first queue) is determined by the number of available services in the second queue. The second queue, Queue 2 in the figure, is a $G/M/\infty$ queue that models the lifetime of a service.

Before starting the simulation, since I did not have a distribution for matching times (μ_0 in the figure), I ran my prototype SE program with a two-level ontology with 36 different types in each level. To get a better estimate on the matching time, I created 1000 different sets of services each having 150 different service descriptions. This gives me 1000 different values of service time. Whenever an arrival occurred, I measured the I/O matching time for that service and associated this time with the number of available services after the arrival of that service. I collected pairs (matching time, number of services) for 150 service

arrivals to use in my simulation. I limited the number of available services to 150 to keep the running time of my prototype SE program reasonable. (Later in this section, I show how this constraint is removed from the simulation model.)

During the simulation, I used exponentially distributed sojourn times for services in Queue 2 [1] with a rate of 1 service per 300 seconds (μ in Figure 6.7). Selecting any equal arrival/departure rate will also result in the same statistics about matching times. Whenever a new service arrives (i.e. a *birth* event happens), if the total number of services in the system does not exceed 150, I find the corresponding matching time randomly from the 1000 pre-calculated matching times. Whenever a service disappears (i.e. a *death* event happens), if there is an available service in the system, I simply decrement the number of available services.

Queue 2 in this Figure 6.7 is a $G/M/\infty$ queue so the average number of services will be [16]:

$$E(n) = \lambda_2 E(s)$$

where λ_2 is the arrival rate to this queue and $E(s)$ is the average lifetime of a service. If $E(n)$, the average number of available services in Queue 2, is less than 150, there is a very low probability of having drops at Queue 1 (Figure 6.7). Thus, the service arrival rate to Queue 2 will be very close to that of Queue 1. Having $E(s) = 300$, the upper limit of service arrival rate to the system before dropping services will be:

$$\lambda = \frac{E(n)}{E(s)} = 0.5 \; service/sec$$

Using the assumed parameter values (i.e. $\lambda = 0.5$ and $E(s) = 300$) and after running the simulation with a maximum of 150 services for a relatively long period of time (10000 service arrivals of which 1000 was used for warm up), I measured the average service time

[1] I selected an exponential distribution for inter-event times since it can model most arrival/departure times.

Table 6.4: Simulation results to find an I/O compatible sequence assuming a maximum of 150 available services and $\lambda = 0.5$

Metrics	Simulation Values
Avg. No. of services in Queue 2	139.94
Avg. matching time ($1/\mu_1$)	0.48 sec
Avg. queue length of Queue 1	0.15
Avg. waiting time in Queue 1	1.4 sec
Avg. lifetime of a service in Queue 2	299.86 sec
Avg. arrival rate to Queue 2 (λ_2)	0.47
Blocking probability	0.07

(i.e. average time required to do an I/O match), average number of available services, average length of queue that services join before doing I/O matching, average waiting time in the queue, average lifetime of a service, and average arrival rate to Queue 2. The results are shown in Table 6.4. The results shown in this table are based on replicating the simulation 100 times. The average number of available services, as can be seen from the table, is lower than the anticipated 150 services. The reason for this discrepancy is that some services are dropped in Queue 1 and the *effective* arrival rate to Queue 2 is lower than $\lambda = 0.5$ (Table 6.4 shows that this value is 0.47). If the arrival rate is decreased to $\lambda = 0.33$ (an arbitrary value less than 0.5), the average number of available services calculated during simulation will completely match with the anticipated value. Table 6.5 shows the collected statistics from a simulation where $\lambda = 0.33$. Table 6.5 presents that with this low arrival rate, the length of the first queue is zero. This indicates that time spent for service composition is very small when services do not arrive in the system very often.

To further ensure that my simulation was correct and to be able to compare the simulation results to corresponding analytical formulas, I used an exponential distribution with $\mu_1 = 1/0.09 = 11$ (instead of pre-calculated matching times) to generate service times in

Table 6.5: Simulation results to find an I/O compatible sequence assuming a maximum of 150 available services and $\lambda = 0.33$

Metrics	Simulation Values
Avg. No. of services in Queue 2	100.1
Avg. matching time $(1/\mu_1)$	0.09 sec
Avg. queue length of Queue 1	0.0
Avg. waiting time in Queue 1	0.0 sec
Avg. lifetime of a service in Queue 2	299.73 sec
Avg. arrival rate to Queue 2 (λ_2)	0.33
Blocking probability	0.0

Table 6.6: Simulation and analytical results to find I/O compatible sequences assuming a maximum of 150 available services, $\lambda = 0.33$, and an exponential distribution for I/O matching times with $\mu_1 = 11$

Metrics	Simulation Values	Analytical Values
Avg. No. of services in Queue 2	100.1	99
Avg. matching time $(1/\mu_1)$ (sec)	0.09	0.09
Avg. queue length of Queue 1	0.00	0.00
Avg. waiting time in Queue 1 (sec)	0.0	0.0
Avg. lifetime of a service in Queue 2 (sec)	299.73	300
Avg. arrival rate to Queue 2 (λ_2)	0.33	0.33

Queue 1. (Table 6.5 shows that the average service time in Queue 1 is 0.09.) The results of running the simulation with the parameters as before (other than service times in Queue 1 which are from an exponential distribution) are presented in Table 6.6. It can be seen from this table that analytical results confirm the simulation results.

To remove the limit of 150 services in the simulation without having to run the prototype SE program for a very long time to collect matching times, I tried to find a distribution for

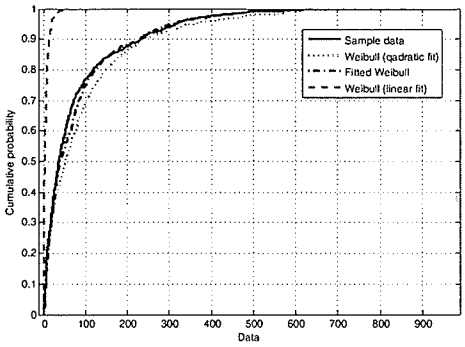
matching times by fitting a distribution to the available sample data. By examining sample matching times when the number of available services was more than 100 (using Matlab's statistics toolbox[2]), I realized that the Weibull distribution [70] might give a good estimate of matching times. I tried different heuristic methods to check the goodness-of-fit of the Weibull distribution for my sample data.

Figure 6.8 shows the cumulative probability function of both the sample data and the estimated Weibull distributions when there are 100, 125, 150 services in the system. (Similar plots were generated for a few other numbers of available services which were similar to the presented figure.) The figure shows the result of Matlab's fitted Weibull distribution (*Fitted Weibull* in the figure) as well as Weibull distributions whose scale and shape parameters are estimated (based on those values from the fitted distribution) using linear and quadratic fits (*Weibull (quadratic fit)* and *Weibull (linear fit)* in the figure). It can be seen that the two plots for sample data and the fitted Weibull are very similar. The estimation is necessary to find scale and shape parameters when there are more than 150 services in the system. The details of estimation will be explained later in this section.

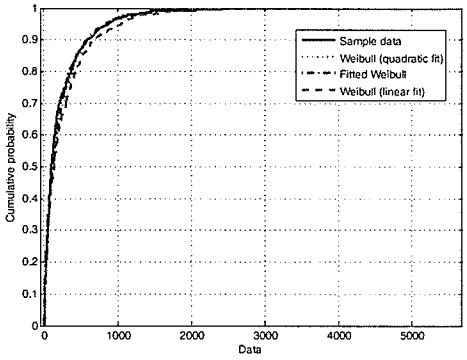
I also tried Q-Q plots [51] when there are 100, 125, and 150 services in the system to see how well the Weibull distribution estimates my sample data. Figure 6.9 shows Q-Q plots of the sample data with the fitted Weibull distribution. The plots are almost linear indicating that the middle of the fitted Weibull distribution agrees with the middle of the true underlying distribution.

For a given number of available services, k, I found the scale parameter, A, and shape parameter, B, of the Weibull distribution. I calculated these two parameters for $k = 100$ to $k = 150$ generating 50 different parameters for the Weibull distribution. Then, I used a polynomial as well as a quadratic fit to extrapolate the parameters of the Weibull distribution for services over 150. Figure 6.10 shows how these two parameters are estimated

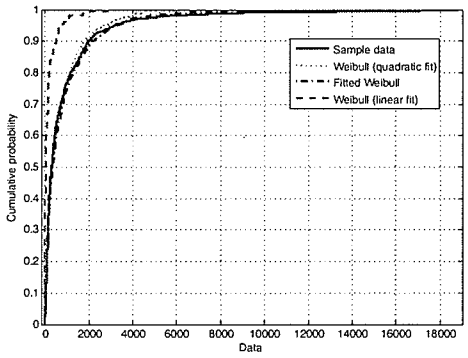
[2]Matlab uses maximum likelihood estimation method to fit a distribution to data.



(a) CDF of sample and the estimated Weibull distributions for $k = 100$



(b) CDF of sample and the estimated Weibull distributions for $k = 125$



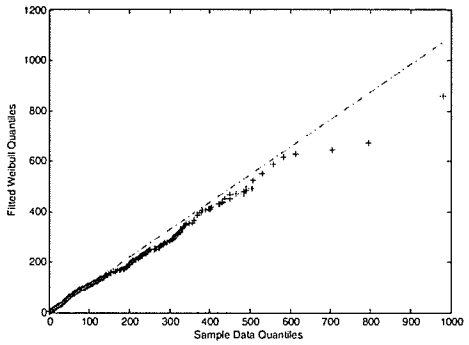
(c) CDF of sample and the estimated Weibull distributions for $k = 150$

Figure 6.8: Cumulative Distribution Function (CDF) of sample data and estimation using Weibull distributions for $k = 100, 125,$ and 150

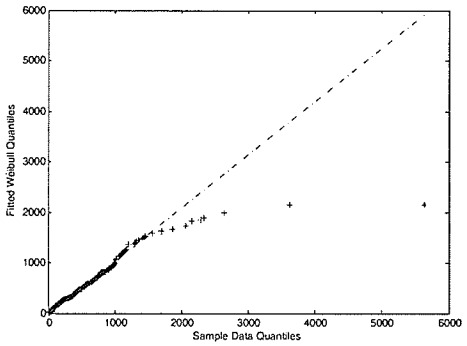
using linear and quadratic fits.

Figure 6.10 suggests using the quadratic fit and the result is:

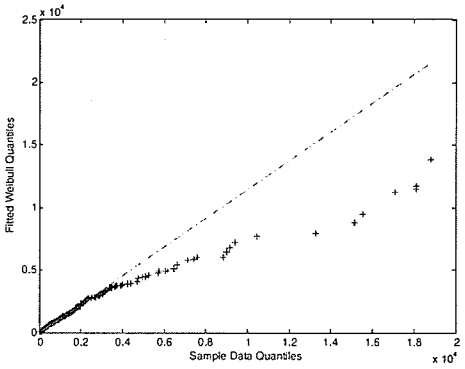
$$\begin{aligned} A &= 0.201k^2 - 41.273k + 2204 \quad \text{where k}\geq 100 \text{ is the number of services} \\ B &= -0.0028k + 1.0096 \end{aligned} \tag{6.1}$$



(a) Q-Q plot of sample data with fitted Weibull distribution for $k = 100$



(b) Q-Q plot of sample data with fitted Weibull distribution for $k = 125$

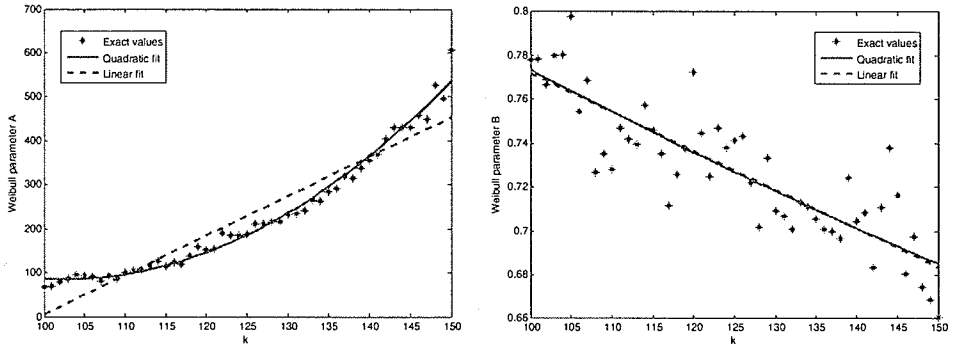


(c) Q-Q plot of sample data with fitted Weibull distribution for $k = 150$

Figure 6.9: Q-Q plots for sample data and the fitted Weibull distribution for $k = 100$, 125, and 150

To further check the suitability of the quadratic fit, I plotted the cumulative distribution function of the Weibull distribution estimated using linear and quadratic methods in Figure 6.8. This figure shows that the quadratic method estimates the fitted Weibull distribution better. I also tried Q-Q plots for quadratic estimate for $k = 100, 125,$ and 150 which are shown in Figure 6.11.

Having parameters for the Weibull distribution, I removed the constraint of 150 services



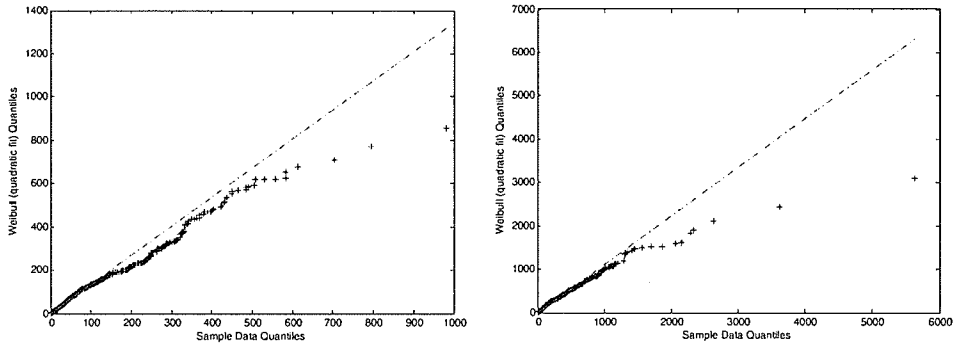
(a) Estimating the scale parameter (b) Estimating the shape parameter

Figure 6.10: Estimating scale and shape parameters of the fitted Weibull distribution using linear and quadratic fits

from my simulation and ran it with the same parameters as before (i.e. $\lambda = 0.5$ and $E(s) = 300$, where $E(s)$ is the sojourn time of a service). If there are less than 100 available services, I randomly chose one of the 1000 sample running times. Otherwise I used formula 6.1 to calculate the parameters of the Weibull distribution for the given number of services, k, and then generated a Weibull random variable as the matching time.

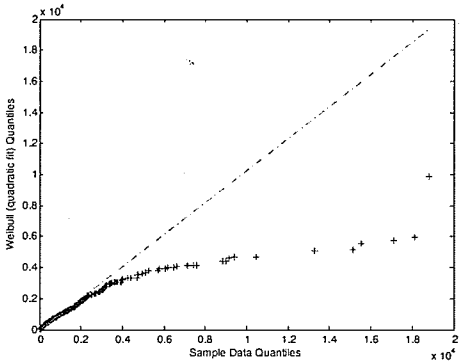
The experiment with the Weibull distribution suggested using a lower arrival rate since the matching times were higher than before. So I selected $\lambda = 0.25$ to keep the system stable. Table 6.7 shows the result of the new simulation experiment. Since I did not drop any services in Queue 1, the arrival rates of both Queue 1 and Queue 2 , as can be seen from Table 6.7, are the same and the average number of available services matches with the anticipated value (i.e. $\lambda \times E(s) = 0.25 \times 300 = 75$).

In the previous simulations, I selected the input and output types of services uniformly from available types. However, this does not reflect a real world scenario in which some of the services are more popular than the others. To model this aspect of the real world, I used Zipf distribution [52] to select the input and output types of services. In the Zipf



(a) Q-Q plot of sample data with quadratic estimated Weibull distribution for $k = 100$

(b) Q-Q plot of sample data with quadratic estimated Weibull distribution for $k = 125$



(c) Q-Q plot of sample data with quadratic estimated Weibull distribution for $k = 150$

Figure 6.11: Q-Q plots for sample data and the quadratic estimated Weibull distribution for $k = 100, 125,$ and 150

distribution, the probability of selecting an element with rank k out of N elements is:

$$f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N} 1/n^s} \text{ where s characterizes the distribution} \tag{6.2}$$

To select the rank of the input and output types, I assume (without loss of generality) that each type is arbitrarily associated with a number between 1 and 72 (recall that we have 72 types in the sample ontology). I use this number as the rank of a type and I choose $s = 1$.

Table 6.7: Simulation results to find an I/O compatible sequence using Weibull distribution without the constraint of having a maximum of 150 available services and $\lambda = 0.25$

Metrics	Simulation Values
Avg. No. of services in Queue 2	73.56
Avg. matching time ($1/\mu_1$)	1.24 sec
Avg. queue length of Queue 1	126
Avg. waiting time in Queue 1	503 sec
Avg. lifetime of a service in Queue 2	299.91 sec
Avg. arrival rate to Queue 2 (λ_2)	0.25

Having these parameters, I can pre-calculate the probability of selecting each element using Formula 6.2. At simulation time, I generated a random number and compared it with the previously calculated Zipf probabilities to select an input and output type for a service. The results of simulating the system using Zipf distribution show only a minor difference in the collected statistics and hence are not provided.

6.4 Scalability Analysis

Having an estimate of the average number of available services at the SE side and the average matching time to find a sequence, I decided to determine how well my system scales. Scalability is an important, practical consideration for SEs. I consider three different environments: 1) a home, 2) a conference/meeting room, and 3) an airport lounge each having different characteristics related to frequency, complexity, etc. of compositions.

The first environment that I chose to analyze, a home, has relatively static behavior in terms of arrival and departure of services. Let us assume that the mean arrival time of a new service to a home is 30 days (I assume a steady period). In other words, on average, every month we will have the arrival of one new service in a home (possibly corresponding

to a somewhat less frequent arrival of new devices providing the new services). Services, once they arrive, will be added to the input/output caches at the SE side for I/O matching. Services will remain in this list for some time and then be removed based on an exponential distribution. The result of I/O matching after ranking, of course, will be added to the permanent repository for future matching. Based on my simulation results, the highest arrival rate before the system becomes unstable (i.e. a system with long waiting times, very long queue length, etc.) is:

$$\lambda_{stable} = 0.25\ service/sec$$

This equation applies to a deterministic system (i.e. the system that I simulated) and just shows an order of magnitude calculation. Having an arrival rate of 1 service each month gives:

$$\lambda_{home} = \frac{1}{30 * 24 * 3600} = 0.386 * 10^{-6}\ service/sec$$

If we assume that each SE can support H GDs (or homes in this case), we will have:

$$\begin{aligned} \lambda_{stable} &= H * \lambda_{home} \\ \Rightarrow max(H) &= \lambda_{stable}/\lambda_{home} = 648 * 10^3\ homes \end{aligned} \tag{6.3}$$

Equation 6.3 suggests that my proposed approach for composition will not cause any problems despite supporting a large number of gateway devices deployed in different homes.

The second environment that I chose to analyze was a conference/meeting room. A conference room is more dynamic than a home in terms of arrival and departure of services. Each time that a speaker changes or a new meeting starts, many arrivals/departures may occur. I assume that changes in a meeting room (and subsequently arrivals/departures) occur based on an exponential distribution with a mean of half an hour. Thus, the arrival

rate will be:

$$\lambda_{meeting} = \frac{1}{30 * 60} = 0.56 * 10^{-3}\ service/sec$$

Using equation 6.3 with the new arrival rate yields:

$$\begin{aligned} \lambda_{stable} &= M * \lambda_{meeting} \\ \Rightarrow max(M) &= \lambda_{stable}/\lambda_{meeting} = 450\ meeting\ rooms \end{aligned}$$

This means that an SE can support a reasonable number of GDs corresponding to different meeting rooms.

The third environment chosen was a very dynamic one, i.e. an airport lounge. If I assume that a wireless access point can cover several gates in an airport, at a moderately busy airport, the service arrival time might follow an exponential distribution with a mean of five minutes. The arrival rate will be:

$$\lambda_{airport} = \frac{1}{5 * 60} = 0.33 * 10^{-2}\ service/sec$$

The maximum number of GDs will be:

$$\begin{aligned} \lambda_{stable} &= L * \lambda_{airport} \\ \Rightarrow max(L) &= \lambda_{stable}/\lambda_{airport} = 75\ lounges \end{aligned}$$

which is a reasonable number.

The matching times used in these analyses came from the simulation study that measured pure I/O matching time. Nevertheless, these results show that even for a very dynamic environment such as an airport lounge, each service enabler is able to support a

reasonable number of GDs. Keeping in mind that after running the proposed service composition system in an environment for a while, most of the compositions can be handled using repository-based matching, the number of GDs that can be supported by each SE, in practice, will be higher than my calculated values. Thus, the scalability of my approach to composition will not be an issue for implementation in a real environment.

Chapter 7

Conclusion and Future Work

Service Oriented Architecture (SOA) is rapidly gaining popularity in developing new software systems. However, most of work has been done in the Web services area and little work has tried to apply this idea in pervasive environments. Improvements in technology have enabled end-user devices (e.g. cell phones, PDAs, etc.) to be powerful enough (from both the computation and memory perspective) to offer services and perform calculations. Having a multitude of pervasive devices/services in an environment gives us the potential opportunity to build and offer new services based on the existing ones. This is highly desirable but users in pervasive environments are seldom capable or interested in overseeing this process.

Most existing service composition systems require a pre-specified "target" workflow and then try to identify available services in a pervasive environment that can be combined to implement that workflow. Unfortunately, describing a target composite service for most users of pervasive environments is a challenging task. Users in such environments do not want to be involved in configuring, running, and maintaining available services. Furthermore, services in pervasive environments, due to the mobility of most of the devices/users,

may come and go frequently. This makes it difficult to create dependable composite services.

In my thesis, I designed, prototyped and evaluated a composition system which can offer new services in pervasive environments containing many heterogeneous (and possibly mobile) devices without involving end-users. Services are described semantically using a domain ontology developed in the thesis. Semantic description enables finding input/output compatible services and generating composite services on the fly. To avoid involving end-users, I assume existing third party SEs who have both the technical knowledge and computing power needed to create new services for end-users. Service enablers maintain a shared repository of previously generated and used composite services to increase reusability and improve the performance of composition by avoiding the expensive process of doing I/O matching. Composite services generated by the SE using input/output matching are ranked to filter out composite services that are unlikely to be of end-user interest. Several criteria based on general forms of weakness in composition are used to rank a composite service. The SE also uses service usage information collected from different pervasive environments to identify "similar" environments. This information can later be used to adjust the computed rank of composite services per environment type. The final stage in a fully automated service composition process is deployment of generated composite services. To make deployment of newly created services seamless, I assume the existence of a GD which provides network connection between pervasive devices/services and the SEs. The GD advertises the received composite services in terms of available pervasive protocols. In this way, the existing devices in such an environment can easily interact with the advertised services.

I implemented a proof of concept prototype running on a PC acting as GD. Different emulated devices used in this prototype provide a variety of services. The GD detects the presence of a new service and interacts with the SE which applies repository based and, if

necessary, input/output based matching to suggest different composite services that can be created involving the newly discovered service.

My implemented prototype has been tested with real-world scenarios to make sure that it is capable of generating composite services using a real domain ontology. It has also been evaluated using different parameters (corresponding to characteristics of different pervasive environments such as a home, meeting/conference room, etc.) to assess the scalability of the service composition model. The results show that repository-based (a.k.a. template-based) service matching is highly scalable. However repository-based matching alone cannot find compositions that do not exist in the repository. On the other hand, interface-based matching incurs more overhead for composition but can find entirely new composite services. Thus, interface-based matching is used to complement the template-based matching technique. To reduce the composition time for future compositions, the results of interface-based matching, after ranking, are stored in the repository. Assuming that SEs provide services to many pervasive environments, the probability of finding suitable compositions in the repository should be very high, making the overall system efficient.

7.1 Contributions

The contributions of my thesis are as follows:

1. A novel fully automatic service composition model for pervasive environments in general and Home Area Networks (HANs) in particular which involves SEs in the service composition process

2. A novel method for storing workflows (i.e. shared repository by the SE) and an integrated and efficient method of semantic matching including optimized both input/output-based and repository-based approaches.

3. Creation of a realistic type ontology used to describe services semantically.

4. A method for ranking the automatically generated composite services (both abstract and concrete) to ensure that only the most useful generated services will be offered to the end-user and thereby avoiding "information overload".

5. Basic verification of the ranking scheme

6. A mechanism for deploying composite services in pervasive environments using OSGi and two available ubiquitous computing protocols (i.e. UPnP and Jini).

7. A novel OSGi grounding for OWL-S descriptions and a mechanism for the dynamic creation of such groundings.

8. A simulation study and scalability analysis to assess the feasibility of supporting many GDs by SEs.

7.2 Future Work

The work presented in this thesis might be extended in the following ways:

- A more complicated ranking function could be developed which would consider parameters like contextual information. Heuristic techniques from Artificial Intelligence (AI) can result in developing a better ranking function. Applying such a ranking function, could select only the most useful services (based on a specific user's contextual information) which could then be deployed in the user's environment.

- Adding more semantic information should result in finding more accurate and relevant compositions. In addition to type information, which is currently used in my service composition model, task/method information could also be used. Task/method

information would provide a higher level of abstraction by which more relevant compositions could be generated. Further, non-functional characteristics (sometimes referred to as QoS parameters) could also be used to refine the composition process. Such characteristics could be supported within the current OWL-S prototype using "properties" associated with each service.

- Handling service failure for composite services is a crucial issue. Since a composite service involves other services (composite or atomic) the likelihood of a composite service failing is higher than an atomic one. In this thesis I touched upon this issue and proposed a simple fault tolerance method. However, saving the partial execution state of a composite service using checkpointing mechanisms might improve the execution reliability of a composite service.

- I did a simulation based assessment of my prototype implementation that only focused on finding average matching time to measure the scalability of the system. Numerous other aspects of the system's use might also be of interest. To study different aspects of a pervasive environment (e.g. response time, overhead of composition, etc.), a (simulation) model must be specifically designed for pervasive environments. The model could then be later used to study the effect of changing different parameters (e.g. arrival rate, service time, etc.) in different pervasive environments.

- Each generated composite service could be conveniently presented to the end-user by having a *widget* or other user-interface object added to their interaction interface. Additionally, each such user-interface could be further customized for each user (e.g. based on user preferences, past usage history, level of user expertise, etc.) to help the end-user to select his/her preferred services easier.

- Modeling pervasive environments based on their device information and usage pattern of composite services can be used to cluster *similar* environments. I assumed a predefined categorization for GDs in my thesis but the same idea might be expanded to automatically identify similar environments. A generated composite service could then be selectively sent to only particular pervasive environments.

- The usage-based ranking implemented in the prototype needs to be tested in an actual deployment to assess the usefulness of this ranking scheme.

Appendix A

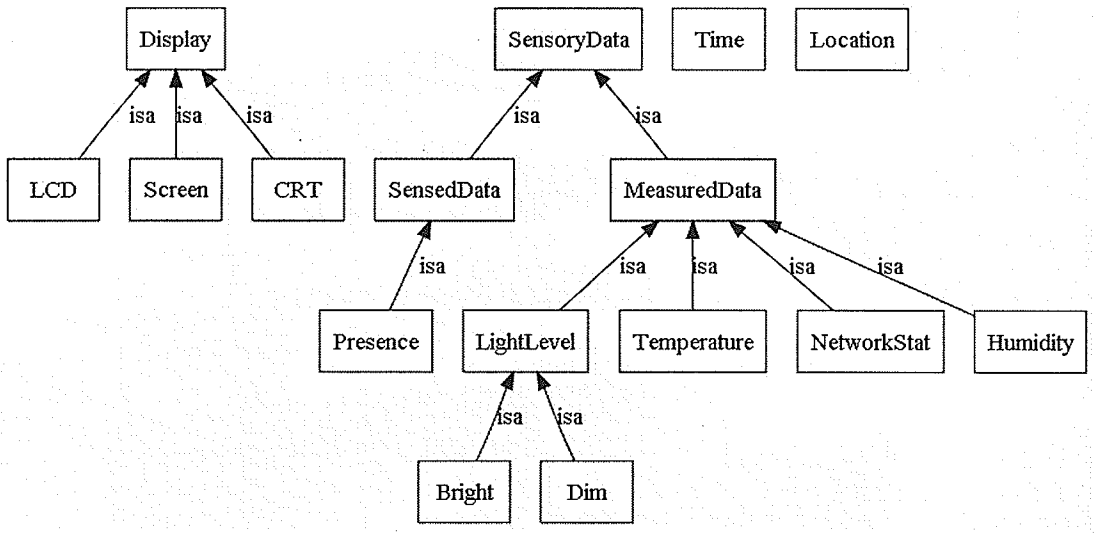
Acronyms

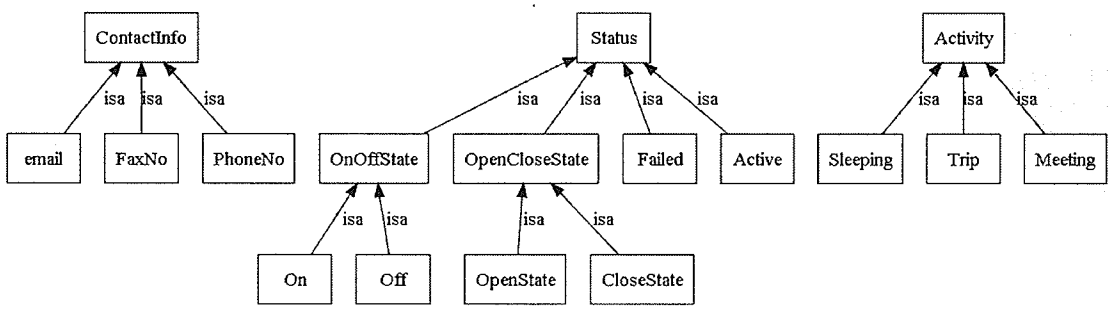
DAML	DARPA Agent Markup Language
DAML-S	DARPA Agent Markup Language for Services
GD	Gateway Device
HAN	Home Area Network
OSGi	Open Service Gateway initiative
OWL	Web Ontology Language
OWL-S	Ontology Web Language for Services
RDF	Resource Description Framework
SDP	Service Discovery Protocol
SE	Service Enabler
SLP	Service Location Protocol
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SOC	Service Oriented Computing
SOUPA	Standard Ontology for Ubiquitous and Pervasive Applications
UDDI	Universal Description Discovery and Integration
UNSPSC	United Nations Standard Products and Services Code
UPnP	Universal Plug and Play
WAN	Wide Area Network
WSDL	Web Service Definition Language

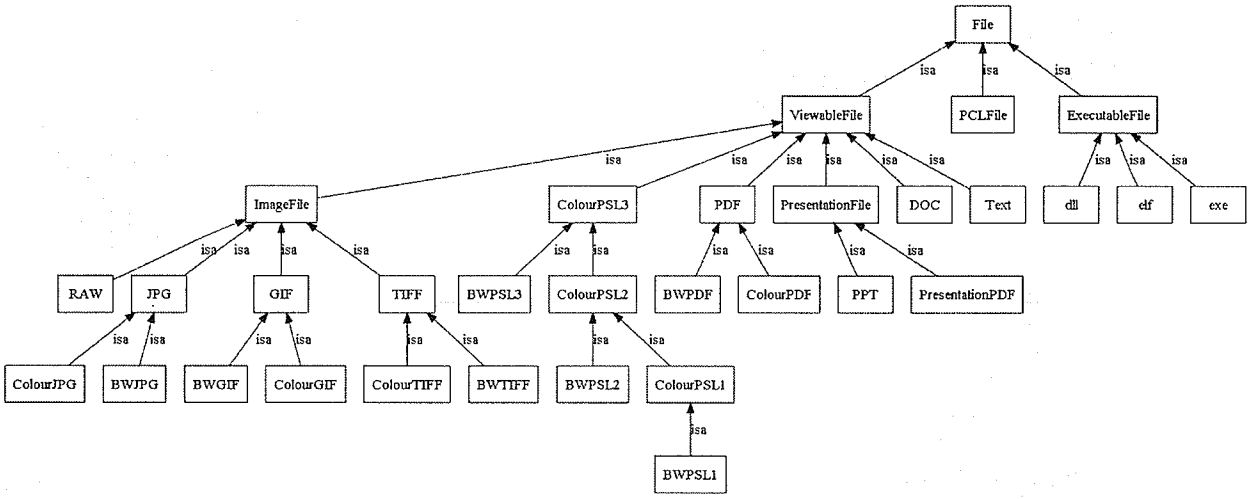
Appendix B

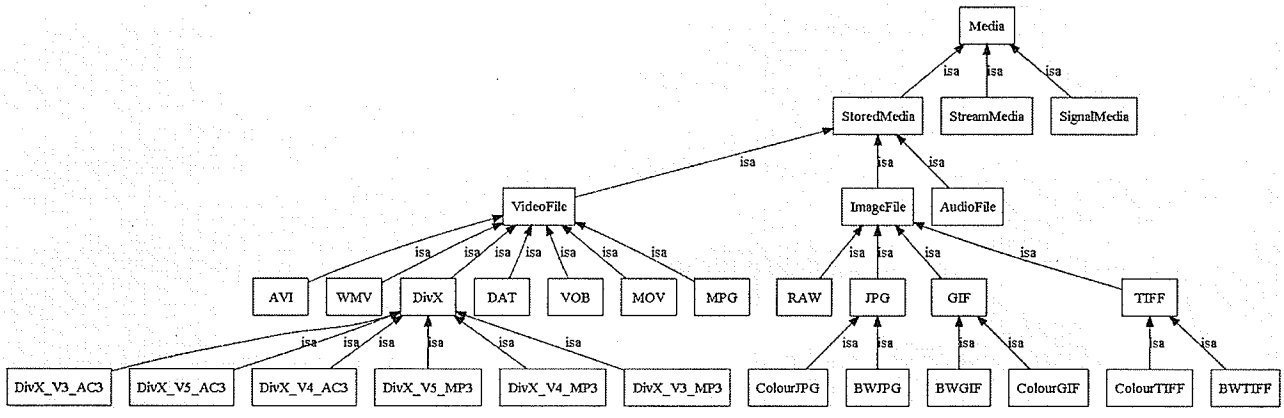
Domain Ontology

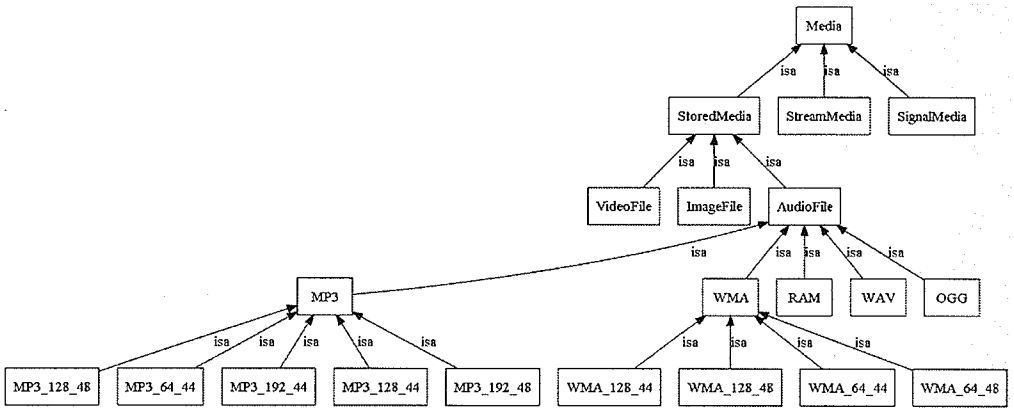
In this appendix I present different parts of the domain ontology designed in this thesis.

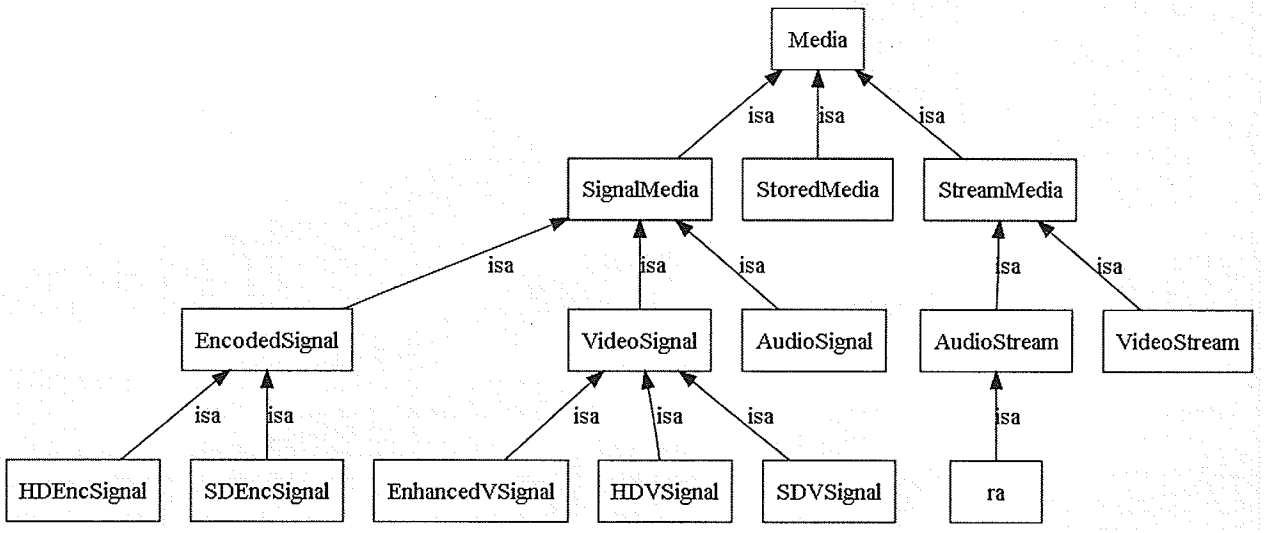



ContactInfo
Status
Activity
email
FaxNo
PhoneNo
OnOffState
OpenCloseState
Failed
Active
Sleeping
Trip
Meeting
On
Off
OpenState
CloseState
isa


File
ViewableFile
PCLFile
ExecutableFile
ImageFile
ColourPSL3
PDF
PresentationFile
DOC
Text
dll
elf
exe
RAW
JPG
GIF
TIFF
BWPSL3
ColourPSL2
BWPDF
ColourPDF
PPT
PresentationPDF
ColourJPG
BWJPG
BWGIF
ColourGIF
ColourTIFF
BWTIFF
BWPSL2
ColourPSL1
BWPSL1
isa


Media
StoredMedia
StreamMedia
SignalMedia
VideoFile
ImageFile
AudioFile
AVI
WMV
DivX
DAT
VOB
MOV
MPG
RAW
JPG
GIF
TIFF
DivX_V3_AC3
DivX_V5_AC3
DivX_V4_AC3
DivX_V5_MP3
DivX_V4_MP3
DivX_V3_MP3
ColourJPG
BWJPG
BWGIF
ColourGIF
ColourTIFF
BWTIFF
isa


Media
StoredMedia
StreamMedia
SignalMedia
VideoFile
ImageFile
AudioFile
MP3
WMA
RAM
WAV
OGG
MP3_128_48
MP3_64_44
MP3_192_44
MP3_128_44
MP3_192_48
WMA_128_44
WMA_128_48
WMA_64_44
WMA_64_48
isa


Media
SignalMedia
StoredMedia
StreamMedia
EncodedSignal
VideoSignal
AudioSignal
AudioStream
VideoStream
HDEncSignal
SDEncSignal
EnhancedVSignal
HDVSignal
SDVSignal
ra
isa

Appendix C

Service Description

Description of a few example services encoded in OWL-S with OSGi grounding are presented in this appendix.

C.1 Display Video and Play Audio Service

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:process="http://www.daml.org/services/owl-s/1.1/Process.owl#"
  xmlns:list="http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl#"
  xmlns:swrl="http://www.w3.org/2003/11/swrl#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:expression="http://www.daml.org/services/owl-s/1.1/generic/
  Expression.owl#"
  xmlns:OE="http://localhost/OWLSExtensions.owl#"
  xmlns="http://localhost/home/DisplayVideoAudio.owl"
```

```
  xmlns:service="http://www.daml.org/services/owl-s/1.1/Service.owl#"
  xmlns:grounding="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:profile="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
  xml:base="http://localhost/home/DisplayVideoAudio.owl">
  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/
Service.owl"/>
    <owl:imports rdf:resource="home-ont.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/
Process.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/
Grounding.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/
Profile.owl"/>
  </owl:Ontology>
  <service:Service rdf:ID="DisplayVideoAudioService">
    <service:presents rdf:resource="#DisplayVideoAudioProfile"/>
    <service:describedBy rdf:resource="#DisplayVideoAudioProcess"/>
    <service:supports rdf:resource="#DisplayVideoAudioGrounding">
    <rdfs:label>DisplayVideoAudioServicelabel</rdfs:label>
  </service:Service>
  <profile:Profile rdf:ID="#DisplayVideoAudioProfile">
    <service:presentedBy rdf:resource="#DisplayVideoAudioService"/>
    <profile:serviceName>DisplayVideoAudioService</profile:serviceName>
    <profile:hasInput rdf:resource="#input1"/>
    <profile:hasInput rdf:resource="#input2"/>
  </profile:Profile>
```

```
  <process:AtomicProcess rdf:ID="#DisplayVideoAudioProcess">
    <service:describes rdf:resource="#DisplayVideoAudioService"/>
    <process:hasInput rdf:resource="#input1"/>
    <process:hasInput rdf:resource="#input2"/>
  </process:AtomicProcess>
  <process:Input rdf:ID="input1">
    <rdfs:label>input1</rdfs:label>
    <process:parameterType rdf:datatype="http://www.w3.org/2001/
XMLSchema#anyURI"
    >http://localhost/home/home-ont.owl#VideoSignal</process:parameterType>
  </process:Input>
  <process:Input rdf:ID="input2">
    <rdfs:label>input2</rdfs:label>
    <process:parameterType rdf:datatype="http://www.w3.org/2001/
XMLSchema#anyURI"
    >http://localhost/home/home-ont.owl#AudioSignal</process:parameterType>
  </process:Input>
  <OE:OSGiGrounding rdf:ID="DisplayVideoAudioGrounding">
    <service:supportedBy rdf:resource="#DisplayVideoAudioService"/>
    <grounding:hasAtomicProcessGrounding
rdf:resource="#DisplayVideoAudioProcessGrounding"/>>
  </OE:OSGiGrounding>
  <OE:OSGiAtomicProcessGrounding rdf:ID="DisplayVideoAudioProcessGrounding">
    <OE:osgiService>DisplayVideoAudio</OE:osgiService>
    <OE:serviceInput>
  <OE:osgiParameter rdf:ID="In0">
        <OE:owlsParameter>
           <process:Input rdf:resource="#input1"/>
        </OE:owlsParameter>
        <OE:javaType>java.lang.String</OE:javaType>
        <OE:paramIndex>0</OE:paramIndex>
```

```
      </OE:osgiParameter>
    </OE:serviceInput>
    <OE:serviceInput>
  <OE:osgiParameter rdf:ID="In1">
        <OE:owlsParameter>
           <process:Input rdf:resource="#input2"/>
        </OE:owlsParameter>
        <OE:javaType>java.lang.String</OE:javaType>
        <OE:paramIndex>1</OE:paramIndex>
      </OE:osgiParameter>
    </OE:serviceInput>
    <OE:osgiClass>org.osgi.service.upnp.UPnPDevice</OE:osgiClass>
    <grounding:owlsProcess>
      <process:AtomicProcess rdf:about="#DisplayVideoAudioProcess"/>
    </grounding:owlsProcess>
  </OE:OSGiAtomicProcessGrounding>
</rdf:RDF>
```

C.2 Play MPG File Service

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:process="http://www.daml.org/services/owl-s/1.1/Process.owl#"
  xmlns:list="http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl#"
  xmlns:swrl="http://www.w3.org/2003/11/swrl#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
```

```
  xmlns:expression="http://www.daml.org/services/owl-s/1.1/generic/
  Expression.owl#"
  xmlns:OE="http://localhost/OWLSExtensions.owl#"
  xmlns="http://localhost/home/PlayMPGFile.owl"
  xmlns:service="http://www.daml.org/services/owl-s/1.1/Service.owl#"
  xmlns:grounding="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:profile="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
  xml:base="http://localhost/home/PlayMPGFile.owl">
  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/
Service.owl"/>
    <owl:imports rdf:resource="home-ont.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/
Process.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/
Grounding.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/
Profile.owl"/>
  </owl:Ontology>
  <service:Service rdf:ID="PlayMPGFileService">
    <service:presents rdf:resource="#PlayMPGFileProfile"/>
    <service:describedBy rdf:resource="#PlayMPGFileProcess"/>
    <service:supports rdf:resource="#PlayMPGFileGrounding">
    <rdfs:label>PlayMPGFileServicelabel</rdfs:label>
  </service:Service>
  <profile:Profile rdf:ID="#PlayMPGFileProfile">
    <service:presentedBy rdf:resource="#PlayMPGFileService"/>
```

```
    <profile:serviceName>PlayMPGFileService</profile:serviceName>
    <profile:hasOutput rdf:resource="#output1"/>
    <profile:hasOutput rdf:resource="#output2"/>
  </profile:Profile>
  <process:AtomicProcess rdf:ID="#PlayMPGFileProcess">
    <service:describes rdf:resource="#PlayMPGFileService"/>
    <process:hasOutput rdf:resource="#output1"/>
    <process:hasOutput rdf:resource="#output2"/>
  </process:AtomicProcess>
  <process:Output rdf:ID="output1">
    <rdfs:label>output1</rdfs:label>
    <process:parameterType rdf:datatype="http://www.w3.org/2001/
XMLSchema#anyURI"
    >http://localhost/home/home-ont.owl#VideoSignal</process:parameterType>
  </process:Output>
  <process:Output rdf:ID="output2">
    <rdfs:label>output2</rdfs:label>
    <process:parameterType rdf:datatype="http://www.w3.org/2001/
XMLSchema#anyURI"
    >http://localhost/home/home-ont.owl#AudioSignal</process:parameterType>
  </process:Output>
  <OE:OSGiGrounding rdf:ID="PlayMPGFileGrounding">
    <service:supportedBy rdf:resource="#PlayMPGFileService"/>
    <grounding:hasAtomicProcessGrounding
rdf:resource="#PlayMPGFileProcessGrounding"/>>
  </OE:OSGiGrounding>
  <OE:OSGiAtomicProcessGrounding rdf:ID="PlayMPGFileProcessGrounding">
    <OE:osgiService>PlayMPGFile</OE:osgiService>
    <OE:serviceOutput>
  <OE:osgiVariable rdf:ID="Out0">
        <OE:owlsParameter>
```

```
            <process:Output rdf:resource="#output1"/>
         </OE:owlsParameter>
         <OE:javaType>java.lang.String</OE:javaType>
 <OE:varIndex>0</OE:varIndex>
      </OE:osgiVariable>
    </OE:serviceOutput>
    <OE:serviceOutput>
  <OE:osgiVariable rdf:ID="Out2">
         <OE:owlsParameter>
            <process:Output rdf:resource="#output2"/>
         </OE:owlsParameter>
         <OE:javaType>java.lang.String</OE:javaType>
 <OE:varIndex>1</OE:varIndex>
      </OE:osgiVariable>
    </OE:serviceOutput>
    <OE:osgiClass>org.osgi.service.upnp.UPnPDevice</OE:osgiClass>
    <grounding:owlsProcess>
      <process:AtomicProcess rdf:about="#PlayMPGFileProcess"/>
    </grounding:owlsProcess>
  </OE:OSGiAtomicProcessGrounding>
</rdf:RDF>
```

C.3 Convert PDF to PS Service

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:process="http://www.daml.org/services/owl-s/1.1/Process.owl#"
  xmlns:list="http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl#"
  xmlns:swrl="http://www.w3.org/2003/11/swrl#"
```

```
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:expression="http://www.daml.org/services/owl-s/1.1/generic/
   Expression.owl#"
  xmlns:OE="http://localhost/OWLSExtensions.owl#"
  xmlns="http://localhost/home/PDFToPS.owl"
  xmlns:service="http://www.daml.org/services/owl-s/1.1/Service.owl#"
  xmlns:grounding="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:profile="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
  xml:base="http://localhost/home/PDFToPS.owl">
  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/
Service.owl"/>
    <owl:imports rdf:resource="home-ont.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/
Process.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/
Grounding.owl"/>
    <owl:imports rdf:resource="http://www.daml.org/services/owl-s/1.1/
Profile.owl"/>
  </owl:Ontology>
  <service:Service rdf:ID="PDFToPSService">
    <service:presents rdf:resource="#PDFToPSProfile"/>
    <service:describedBy rdf:resource="#PDFToPSProcess"/>
    <service:supports rdf:resource="#PDFToPSGrounding">
    <rdfs:label>PDFToPSServicelabel</rdfs:label>
  </service:Service>
```

```
  <profile:Profile rdf:ID="#PDFToPSProfile">
    <service:presentedBy rdf:resource="#PDFToPSService"/>
    <profile:serviceName>PDFToPSService</profile:serviceName>
    <profile:hasInput rdf:resource="#input1"/>
    <profile:hasOutput rdf:resource="#output1"/>
  </profile:Profile>
  <process:AtomicProcess rdf:ID="#PDFToPSProcess">
    <service:describes rdf:resource="#PDFToPSService"/>
    <process:hasInput rdf:resource="#input1"/>
    <process:hasOutput rdf:resource="#output1"/>
  </process:AtomicProcess>
  <process:Input rdf:ID="input1">
    <rdfs:label>input1</rdfs:label>
    <process:parameterType rdf:datatype="http://www.w3.org/2001/
XMLSchema#anyURI"
    >http://localhost/home/home-ont.owl#pdf</process:parameterType>
  </process:Input>
  <process:Output rdf:ID="output1">
    <rdfs:label>output1</rdfs:label>
    <process:parameterType rdf:datatype="http://www.w3.org/2001/
XMLSchema#anyURI"
    >http://localhost/home/home-ont.owl#ColourPSL3</process:parameterType>
  </process:Output>
  <OE:OSGiGrounding rdf:ID="PDFToPSGrounding">
    <service:supportedBy rdf:resource="#PDFToPSService"/>
    <grounding:hasAtomicProcessGrounding
rdf:resource="#PDFToPSProcessGrounding"/>>
  </OE:OSGiGrounding>
  <OE:OSGiAtomicProcessGrounding rdf:ID="PDFToPSProcessGrounding">
    <OE:osgiService>PDFToPS</OE:osgiService>
    <OE:serviceInput>
```

```
 <OE:osgiParameter rdf:ID="In0">
        <OE:owlsParameter>
           <process:Input rdf:resource="#input1"/>
        </OE:owlsParameter>
        <OE:javaType>java.lang.String</OE:javaType>
<OE:paramIndex>0</OE:paramIndex>
     </OE:osgiParameter>
   </OE:serviceInput>
   <OE:serviceOutput>
 <OE:osgiVariable rdf:ID="Out0">
        <OE:owlsParameter>
           <process:Output rdf:resource="#output1"/>
        </OE:owlsParameter>
        <OE:javaType>java.lang.String</OE:javaType>
<OE:varIndex>0</OE:varIndex>
     </OE:osgiVariable>
   </OE:serviceOutput>
   <OE:osgiClass>org.osgi.service.upnp.UPnPDevice</OE:osgiClass>
   <grounding:owlsProcess>
     <process:AtomicProcess rdf:about="#PDFToPSProcess"/>
   </grounding:owlsProcess>
 </OE:OSGiAtomicProcessGrounding>
</rdf:RDF>
```

Appendix D

Additional Service Composition Scenarios

Results of Scenario1:

```
rank:  0.59   DecodeSignalServicePlayDivXFileService--
              DisplayVideoAudioService
rank:  0.59   DecodeSignalServicePlayDVDService--
              DisplayVideoAudioService
rank:  0.59   DecodeSignalServicePlayMPGFileService--
              DisplayVideoAudioService
rank:  0.59   DecodeSignalServicePlayWMVFileService--
              DisplayVideoAudioService
rank:  0.59   PlayDivXFileServiceDecodeSignalService--
              DisplayVideoAudioService
rank:  0.59   PlayDivXFileServicePlayDVDService--
              DisplayVideoAudioService
```

```
rank:  0.59   PlayDivXFileServicePlayMPGFileService--
              DisplayVideoAudioService
rank:  0.59   PlayDivXFileServicePlayWMVFileService--
              DisplayVideoAudioService
rank:  0.59   PlayDVDServiceDecodeSignalService--
              DisplayVideoAudioService
rank:  0.59   PlayDVDServicePlayDivXFileService--
              DisplayVideoAudioService
rank:  0.59   PlayDVDServicePlayMPGFileService--
              DisplayVideoAudioService
rank:  0.59   PlayDVDServicePlayWMVFileService--
              DisplayVideoAudioService
rank:  0.59   PlayMPGFileServiceDecodeSignalService--
              DisplayVideoAudioService
rank:  0.59   PlayMPGFileServicePlayDivXFileService--
              DisplayVideoAudioService
rank:  0.59   PlayMPGFileServicePlayDVDService--
              DisplayVideoAudioService
rank:  0.59   PlayMPGFileServicePlayWMVFileService--
              DisplayVideoAudioService
rank:  0.59   PlayWMVFileServiceDecodeSignalService--
              DisplayVideoAudioService
rank:  0.59   PlayWMVFileServicePlayDivXFileService--
              DisplayVideoAudioService
rank:  0.59   PlayWMVFileServicePlayDVDService--
              DisplayVideoAudioService
```

```
rank:  0.59   PlayWMVFileServicePlayMPGFileService--
              DisplayVideoAudioService
rank:  0.7    Composite_RecordMP3FileServicelabel_PlayMP3_64_44File
            ServicelabelDecodeSignalService--DisplayVideoAudioService
rank:  0.7    Composite_RecordMP3FileServicelabel_PlayMP3_64_44File
            ServicelabelPlayDivXFileService--DisplayVideoAudioService
rank:  0.7    Composite_RecordMP3FileServicelabel_PlayMP3_64_44File
            ServicelabelPlayDVDService--DisplayVideoAudioService
rank:  0.7    Composite_RecordMP3FileServicelabel_PlayMP3_64_44File
            ServicelabelPlayMPGFileService--DisplayVideoAudioService
rank:  0.7    Composite_RecordMP3FileServicelabel_PlayMP3_64_44File
            ServicelabelPlayWMVFileService--DisplayVideoAudioService
rank:  0.73   DecodeSignalServicePlayBWJPGFileService--
              DisplayVideoAudioService
rank:  0.73   DecodeSignalServicePlayColourJPGFileService--
              DisplayVideoAudioService
rank:  0.73   PlayDivXFileServicePlayBWJPGFileService--
              DisplayVideoAudioService
rank:  0.73   PlayDivXFileServicePlayColourJPGFileService--
              DisplayVideoAudioService
rank:  0.73   PlayDVDServicePlayBWJPGFileService--
              DisplayVideoAudioService
rank:  0.73   PlayDVDServicePlayColourJPGFileService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_128_44FileServiceDecodeSignalService--
              DisplayVideoAudioService
```

```
rank:  0.73   PlayMP3_128_44FileServicePlayDivXFileService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_128_44FileServicePlayDVDService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_128_44FileServicePlayMPGFileService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_128_44FileServicePlayWMVFileService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_128_48FileServiceDecodeSignalService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_128_48FileServicePlayDivXFileService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_128_48FileServicePlayDVDService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_128_48FileServicePlayMPGFileService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_128_48FileServicePlayWMVFileService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_192_44FileServiceDecodeSignalService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_192_44FileServicePlayDivXFileService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_192_44FileServicePlayDVDService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_192_44FileServicePlayMPGFileService--
              DisplayVideoAudioService
```

```
rank:  0.73   PlayMP3_192_44FileServicePlayWMVFileService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_192_48FileServiceDecodeSignalService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_192_48FileServicePlayDivXFileService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_192_48FileServicePlayDVDService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_192_48FileServicePlayMPGFileService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_192_48FileServicePlayWMVFileService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_64_44FileServiceDecodeSignalService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_64_44FileServicePlayDivXFileService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_64_44FileServicePlayDVDService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_64_44FileServicePlayMPGFileService--
              DisplayVideoAudioService
rank:  0.73   PlayMP3_64_44FileServicePlayWMVFileService--
              DisplayVideoAudioService
rank:  0.73   PlayMPGFileServicePlayBWJPGFileService--
              DisplayVideoAudioService
rank:  0.73   PlayMPGFileServicePlayColourJPGFileService--
              DisplayVideoAudioService
```

```
rank:  0.73   PlayWMVFileServicePlayBWJPGFileService--
              DisplayVideoAudioService
rank:  0.73   PlayWMVFileServicePlayColourJPGFileService--
              DisplayVideoAudioService
rank:  0.8    DecodeSignalService--PlayAudioService
rank:  0.8    PlayDivXFileService--PlayAudioService
rank:  0.8    PlayDVDService--PlayAudioService
rank:  0.8    PlayMPGFileService--PlayAudioService
rank:  0.8    PlayWMVFileService--PlayAudioService
rank:  0.88   Composite_RecordMP3FileServicelabel_PlayMP3_64_44File
              ServicelabelPlayBWJPGFileService--
              DisplayVideoAudioService
rank:  0.88   Composite_RecordMP3FileServicelabel_PlayMP3_64_44File
              ServicelabelPlayColourJPGFileService--
              DisplayVideoAudioService
rank:  0.92   Composite_RecordMP3FileServicelabel_PlayMP3_64_44File
              Servicelabel--PlayAudioService
rank:  0.92   PlayMP3_128_44FileServicePlayBWJPGFileService--
              DisplayVideoAudioService
rank:  0.92   PlayMP3_128_44FileServicePlayColourJPGFileService--
              DisplayVideoAudioService
rank:  0.92   PlayMP3_128_48FileServicePlayBWJPGFileService--
              DisplayVideoAudioService
rank:  0.92   PlayMP3_128_48FileServicePlayColourJPGFileService--
              DisplayVideoAudioService
rank:  0.92   PlayMP3_192_44FileServicePlayBWJPGFileService--
```

```
                DisplayVideoAudioService
rank:   0.92    PlayMP3_192_44FileServicePlayColourJPGFileService--
                DisplayVideoAudioService
rank:   0.92    PlayMP3_192_48FileServicePlayBWJPGFileService--
                DisplayVideoAudioService
rank:   0.92    PlayMP3_192_48FileServicePlayColourJPGFileService--
                DisplayVideoAudioService
rank:   0.92    PlayMP3_64_44FileServicePlayBWJPGFileService--
                DisplayVideoAudioService
rank:   0.92    PlayMP3_64_44FileServicePlayColourJPGFileService--
                DisplayVideoAudioService
rank:   1.0   DecodeSignalService--DisplayVideoAudioService
rank:   1.0   PlayDivXFileService--DisplayVideoAudioService
rank:   1.0   PlayDVDService--DisplayVideoAudioService
rank:   1.0   PlayMP3_128_44FileService--PlayAudioService
rank:   1.0   PlayMP3_128_48FileService--PlayAudioService
rank:   1.0   PlayMP3_192_44FileService--PlayAudioService
rank:   1.0   PlayMP3_192_48FileService--PlayAudioService
rank:   1.0   PlayMP3_64_44FileService--PlayAudioService
rank:   1.0   PlayMPGFileService--DisplayVideoAudioService
rank:   1.0   PlayWMVFileService--DisplayVideoAudioService
rank:   1.0   RecordMP3FileService--PlayMP3_64_44FileService
```

It can be seen from the output that higher ranked composite services (e.g. playing a video file on a TV and playing an MP3 file using a TV's speaker/ an amplifier) are more useful than the others.

Table D.1: Services and their input and output types participating in Scenario A

Service Name	Input Type	Output Type
BWPSL1Print	BWPSL1	
ColourJPGToBWPSL1	ColourJPG	BWPSL1
ColourJPGToColourPSL1Service	ColourJPG	ColourPSL1
ColourPSL1Print	ColourPSL1	None
PlayMP3_64_44File	MP3_64_44	AudioSignal
RecordMP3File	None	MP3_64_44

Scenario A: In this scenario I add an MP3 player, a digital camera, and a PostScript printer. Table D.1 shows some of these services and their corresponding input and output types. In this scenario, only three composite services are generated as follows:

```
rank: 1.0   ColourJPGToBWPSL1Service--BWPSL1PrintService
rank: 1.0   ColourJPGToColourPSL1Service--ColourPSL1PrintService
rank: 1.0   RecordMP3FileService--PlayMP3_64_44FileService
```

Scenario B: This scenario shows the interaction of services in a smart meeting room. Services such as "Display Video", "Play Multimedia Presentation", "Set Light Level", etc. participate in this scenario. Table D.2 shows services and their corresponding input and output types.

The result of composition is as follows:

```
rank: 0.73   GetPresentationFileService--
Composite_PlayMultiMediaPresentationService_DisplayVideoService
```

Table D.2: Services and their input and output types participating in Scenario B

Service Name	Input Type	Output Type
DisplayVideo	VideoSignal	None
GetPresentationFile	Location and Time	PresentationFile
PlayAudio	AudioSignal	None
PlayMultiMediaPresentation	PresentationFile	VideoSignal and AudioSignal
PlayPresentationFile	PresentationFile	VideoSignal
SetBlind	OpenCloseState	Status
SetLightLevel	LightLevel	Status

```
rank: 0.73   GetPresentationFileService--
Composite_PlayMultiMediaPresentationService_PlayAudioService
rank: 0.8   PlayMultiMediaPresentationService--DisplayVideoService
rank: 0.8   PlayMultiMediaPresentationService--PlayAudioService
rank: 0.92   PlayMultiMediaPresentationService--
PlayAudioServiceDisplayVideoService
rank: 0.92   GetPresentationFileService--
Composite_PlayPresentationFileService_DisplayVideoService
rank: 1.0   GetPresentationFileService--
PlayMultiMediaPresentationService
rank: 1.0   GetPresentationFileService--
PlayPresentationFileService
rank: 1.0   PlayPresentationFileService--DisplayVideoService
```

Bibliography

[1] Business Process Execution Language for Web Services. http://xml.coverpages.org/bpel4ws.html.

[2] DAML Web site. http://www.daml.org.

[3] Oscar: An OSGi framework implementation. http://oscar.objectweb.org/.

[4] Resource Description Framework (RDF). http://www.w3.org/RDF.

[5] SSJ: Stochastic simulation in Java. http://www.iro.umontreal.ca/ lecuyer/ssj/index.html.

[6] Web services description language (wsdl) 1.1. http://www.w3.org/TR/wsdl.

[7] HAVi, the A/V digital network revolution. http://www.havi.org/pdf/white.pdf, 1999.

[8] UDDI technical white paper. http://www.uddi.org/pubs/Iru_UDDI_Technical _White_Paper.pdf, 2000.

[9] Universal plug and play device architecture. Microsoft, http://www.upnp.org/download/UPnPDA10_20000613.htm, June 2000.

[10] Jini architecture specification version 2.0. Sun Microsystems, http://wwws.sun.com/software/jini/specs/jini2_0.pdf, June 2003.

[11] Obje interoperability framework. http://www.parc.com/research/csl/projects/obje/Obje_Whitepaper.j 2003.

[12] http://www.w3.org/TR/owl-ref/, 2004.

[13] Resource description framework (RDF): Concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts/, February 2004.

[14] V. Agarwal, G. Chafle, K. Dasgupta, N. Karnik, A. Kumar, S. Mittal, and B. Srivastava. Synthy: A system for end to end composition of Web services. *Journal of Web Semantics*, 3(4):311–339, 2005.

[15] M. Aiello. The role of web services at home. In *Proceedings of the International Conference on Internet and Web Applications and Services/Advanced International Conference on Telecommunications (AICT-ICIW'06)*, pages 164–170, 2006.

[16] A. O. Allen. *Probability, statistics, and queueing theory with computer science applications*. Academic Press, 2 edition, 1990.

[17] OSGI Alliance. The Open Services Gateway initiative. http://www.osgi.org.

[18] I. B. Arpinar, B. Aleman-Meza, R. Zhang, and A. Maduko. Ontology-driven Web services composition. In *Proceedings of IEEE International Conference on E-Commerce Technology*, pages 146–152, 2004.

[19] J. Bourcier, A. Chazalet, M. Desertot, C. Escoffier, and C. Marin. A dynamic-SOA home control gateway. In *Proceedings of the IEEE International Conference on Services Computing (SCC'06)*, pages 463–470, 2006.

[20] K. Bowers, K. Mills, and S. Rose. Self-adaptive leasing for Jini. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 539–542, March 2003.

[21] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M. Shan. Adaptive and dynamic service composition in eFlow. Technical Report HPL-2000-39, HP Software Technology Laboratory, 2000.

[22] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M. Shan. eFlow: a platform for developing and managing composite e-services. In *Proceeedings of the Academia/Industry Working Conference onResearch Challenges*, pages 341–348, 2000.

[23] G. Chafle, K. Dasgupta, A. Kumar, S. Mittal, and B. Srivastava. Adaptation in web service composition and execution. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, pages 549–557, 2006.

[24] D. Chakraborty and A. Joshi. Dynamic service composition: State-of-the-art and research directions. Technical Report TR-CS-01-19, University of Maryland, 2001.

[25] D. Chakraborty and A. Joshi. GSD: A novel group-based service discovery protocol for MANETS. In *Proceedings of the 4th IEEE Conference on Mobile and Wireless Communications Networks (MWCN 2002)*, pages 140–144, September 2002.

[26] D. Chakraborty, F. Perich, S. Avancha, and A. Joshi. Dreggie: A smart service discovery technique for E-commerce applications. In *Workshop on Reliable and Secure Applications in Mobile Environment,In Conjunction with 20th Symposium on Reliable Distributed Systems (SRDS)*, 2001.

[27] D. Chakraborty, Y. Yesha, and A. Joshi. A distributed service composition protocol for pervasive environments. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC)*, pages 2575–2580, March 2004.

[28] W. Y. Chen. *Home networking basis: Transmission environment and wired/wireless protocols*. Prentice Hall, 2004.

[29] J. Cohen, S. Aggarwal, and Y. Y. Goland. Internet Draft, general event notification architecture, 2000.

[30] W3C Consortium. OWL-S 1.1 release. http://www.daml.org/services/owl-s/1.1, 2004.

[31] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the Web services Web: an introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, 2002.

[32] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 24–35, 1999.

[33] P. Dobrev, D. Famolari, C. Kurzke, and B. A. Miller. Device and service discovery in home networks with OSGi. *IEEE Communications Magazine*, 40(8):86–92, August 2002.

[34] N. Feng, G. Ao, T. White, and B. Pagurek. Dynamic evolution of network management software by software hot-swapping. In *Proceedings of the IEEE/IFIP Symposium on Integrated Network Management*, pages 63–76, 2001.

[35] D. Fensel, H. Lausen, A. Polleres, J. de Bruijin, M. Stollberg, D. Roman, and J. Domingue. In *Enabling Semantic Web Services*, chapter 3, pages 25–36. Springer Berlin Heidelberg, 2007.

[36] K. Fujii and T. Suda. Semantic-based dynamic service composition. *IEEE Journal on Selected Areas in Communications*, 23(12):2361–2372, 2005.

[37] N. Georgantas, S. B. Mokhtar, Y. Bromberg, V. Issarny, J. Kalaoja, J. Kantarovitch, A. Grodolle, and R. Mevissen. The Amigo service architecture for the open networked home environment. In *Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture*, 2005.

[38] Y. Y. Goland, T. Cai, P. Leach, and Y. Gu. Intenet Draft, simple service discovery protocol/1.0. http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt, 2000.

[39] A. Gorbenko, V. Kharchenko, P. Popov, and A. Romanovsky. Dependable composite Web services with components upgraded online. In *Architecting Dependable Systems*, volume 3549 of *Lecture Notes in Computer Science*, pages 96–128, 2005.

[40] S. D. Gribble and et al. The Ninja architecture for robust Internet-scale systems and services. *Computer Networks, Special Issue on Pervasive Computing*, 35(4):473–497, 2001.

[41] T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, 43(5):907–928, 1995.

[42] X. Gu, K. Nahrstedt, and B. Yu. SpiderNet: An integrated peer-to-peer service composition framework. In *Proceedings of IEEE International Symposium on High-Performance Distributed Computing*, pages 110–119, 2004.

[43] R. Gupta, S. Talwar, and D. P. Agrawal. Jini home networking: a step toward pervasive computing. *IEEE Computer*, 8(35):34–40, August 2002.

[44] E. Guttman. Service location protocol: automatic discovery of IP network services. *IEEE Internet Computing*, 3(4):71–80, 1999.

[45] E. Guttman, C. Perkins, J. Veizades, and M. Day. RFC2608: Service location protocol, version 2. ftp://ftp.isi.edu/in-notes/rfc2608.txt, 1999.

[46] S. V. Hashemian and F. Mavaddat. A graph-based approach to web services composition. In *Proceedings of the International Symposium on Applications and the Internet*, pages 183–189, 2005.

[47] S. Kalasapur, M. Kumar, and B. Shirazi. Personalized service composition for ubiquitous multimedia delivery. In *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, pages 258–263, 2005.

[48] S. Kalasapur, M. Kumar, and B. Shirazi. Seamless service composition SeSCo in pervasive environments. In *Proceedings of the first ACM international workshop on Multimedia Service Composition*, pages 11–20, 2005.

[49] A. Kaminsky. JiniME: Jini^{TM} connection technology for mobile devices. http://www.cs.rit.edu/ anhinga/whitepapers/JiniMEWhitePaper, August 2000.

[50] J. Kardach. Bluetooth architecture overview. http://www.intel.com/technology/itj/q22000/pdf/art_1.pdf.

[51] A. M. Law and W. D. Kelton. *Simulation modeling and analysis*. Mc Graw Hill, 3 edition, 2000.

[52] W. Li. Random texts exhibit Zipf's-law-like word frequency distribution. *IEEE Transactions on Information Theory*, 38(6):1842–1845, 1992.

[53] Y. Liang, H. Bao, and H. Liu. Hybrid ontology integration for distributed system. In *Proceedings of the 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 309–314, 2007.

[54] S. Majithia, D. W. Walker, and W.A.Gray. Automated Web service composition using semantic Web technologies. In *International Conference on Autonomic Computing (ICAC-04)*, pages 306–307, May 2004.

[55] D. Marples and P. Kriens. The open services gateway initiative: An introductory overview. *IEEE Communications Magazine*, 39(12):110–114, December 2001.

[56] R. Masuoka, Y. Labrou, B. Parsia, and E. Sirin. Ontology-enabled pervasive computing applications. *IEEE intelligent Systems*, 18(5):68–72, 2003.

[57] R. Masuoka, B. Parsia, and Y. Labrou. Task Computing - the semantic Web meets pervasive computing. In *Proceedings of the 2nd International Semantic Web Conference (ISWC)*, pages 866–881, 2003.

[58] A. Messer et al. Interplay: a middleware for seamless device integration and task orchestration in a networked home. In *Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications*, pages 296–305, March 2006.

[59] B. A. Miller, T. Nixon, C. Tai, and M. D. Wood. Home networking with universal plug and play. *IEEE Communications Magazine*, 39(12):104–109, December 2001.

[60] E. Newcomer and G. Lomow. *Understanding SOA with Web services*. Addison Wesley, 2005.

[61] Q. Ni. Service composition in ontology enabled service oriented architecture for pervasice computing. A position paper in a joint workshop between the UK-UbiNet Ubiquitous Computing Network and the e-Science Programme, 2005.

[62] G. O'driscoll. *The essential guide to home networking technologies*. Prentice Hall, Upper Saddle River, 2001.

[63] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Semantic matching of Web services capabilities. In *Proceedings of the 1st International Semantic Web Conference (ISWC2002)*, volume 2342 of *Lecture Notes in Computer Science*, pages 333–347. Springer-Verlag, 2002.

[64] M. P. Papazoglou. Service-oriented computing: concepts, characteristics and directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering, WISE 2003*, pages 3–12, 2003.

[65] S. R. Ponnekanti and A. Fox. SWORD: A developer toolkit for web service composition. In *Proceedings of the Eleventh World Wide Web Conference (Web Engineering Track)*, 2002.

[66] S. R. Ponnekanti and A. Fox. Application-service interoperation without standardized service interfaces. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, pages 30–37, March 2003.

[67] A. Rakotonirainy and G. Groves. Resource discovery for pervasive environments. In *International Conferences on Distributed Objects and Applications (DOA)*, volume 2519 of *Lecture Notes in Computer Science*, pages 866–883. Springer-Verlag, 2002.

[68] J. Rao, P. Kungas, and M. Matskin. Logic-based web service composition: from service description to process model. In *Proceedings of the IEEE International Conference on Web Services*, pages 446–453, July 2004.

[69] J. Rao and X. Su. A survey of automated Web service composition methods. In *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition*, 2004.

[70] S. Ross. *Introduction to Probability Models*. Academic Press, ninth edition, 2007.

[71] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[72] D. Saha and A. Mukherjee. Pervasive computing: a paradigm for the 21st century. *IEEE Computer*, 36(3):25–31, 2003.

[73] U. Saif, D. Gordon, and D. Greaves. Internet access to a home area network. *IEEE Internet Computing*, 5(1):54–63, February 2001.

[74] M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8(4):10–17, 2001.

[75] C. Shankar, A. Ranganathan, and R. Campbell. Towards fault tolerant pervasive computing. *IEEE Technology and Society*, 24(1):38–44, 2005.

[76] E. Sirin. The OWL-S API. http://www.mindswap.org/2004/owl-s/api/.

[77] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic composition of web services using semantic descriptions. In *Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003*, pages 17–24, April 2003.

[78] E. Sirin, B. Parsia, and J. Hendler. Filtering and selecting semantic Web services with interactive composition techniques. *IEEE Intelligent Systems*, 19(4):42–49, 2004.

[79] W. Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, second edition, 1998.

[80] V. Sundramoorthy and J. Scholten. Challenges in the at home anywhere (@ha) service discovery protocol. In *7th Cabernet Radicals Workshop*, October 2002.

[81] V. Sundramoorthy, J. Scholten, P. G. Jansen, and P. H. Hartel. Service discovery at home. In *Proceedings of the 4th International Conference on Information, Communications and Signal Processing and 4th IEEE Pacific-Rim Conference On Multimedia (ICICS/PCM)*, pages 1929–1934, 2003.

[82] W3C technical recommendation. Simple object access protocol version 1.2. http://www.w3.org/TR/soap12-part1/, 2001.

[83] V. Tosic, D. Mennie, and B. Pagurek. On dynamic service composition and and its applicability to e-business software systems. In *Proceedings of the workshop on Object-Oriented Business Systems*, pages 95–108, 2001.

[84] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999.

[85] M. Weiser. The computer for the 21st century. *Scientific American*, pages 94–100, 1991.

[86] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for Web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.

[87] F. Zhu, M. Mutka, and L. Ni. Classificationi of service discovery in pervasive computing environments. Technical Report MSU-CSE-02-24, Michigan State University, EastLansing, 2002.