Service Composition in Ontology enabled Service Oriented Architecture for Pervasive Computing

Qun Ni
Department of Computing, Imperial College London
qun.ni@doc.ic.ac.uk

Abstract

One important issue in Pervasive Computing is how to assist non-expert users to use devices and even to combine their functionality without explicit goals. In this paper, we present an Ontology enabled Service Oriented Architecture (OSOA), which combines interoperability provided by Web services and semantic description provided by Ontologies, to solve this issue. We argue that ad-hoc service composition based on OSOA provides a promising solution for the combination of devices without an explicit goal. We also discuss related issues of applying planning to service composition, and present our ideas to solve these issues.

1. Introduction

A typical pervasive Computing environment is full of devices and services and is very dynamic. Typical users are not Computer Scienties and may be easily overwhelmed by numerous new electrical devices and Internet services. Hence, the problem is how to assist users survive in this service rich and dynamic environment.

A possible answer is letting services provided by these devices (like a wireless camera and a PDA), become smart so that smarter software (agents), not people, can automatically searche, exploit, combine, and perform their functionalities according to user's preferences and context, e.g., using the PDA to watch video from the camera over the door to see who is ringing the bell.

2. Scenario

We give three examples to illustrate our approach.

Example 1: Using a PDA to control a HDTV, a Media Center, and any other networked devices, such as channel selection, playlist adjustment etc.

Example 2: Using a PDA to watch video from a security camera. The difference is a user has to use a display device to show the video because the camera itself cannot show anything.

Example 3: Using a PDA to send a X3D file (3D video part) to a Game Console for rendering, which in turn outputs video signal to a HDTV to display the final result. Meanwhile, the PDA decodes the audio part of the X3D file and sends audio signal to a wireless speaker. This allows you to show your new house plans to your friends, which wouldnot be possible if you do not have a 3D workstation at home.

3. Approach

Obviously, the first issue to solve is interoperability between different services provided by devices. By building a self-contained system for description and messaging that is devoid of programming language or operating system details, Web services have shown that it is possible for applications running in truly disparate environments to communicate securely and reliably, which provides a good and secure base for Pervasive Computing solutions. However, it is also clear that Web services cannot be used to solve service composition problem because the interaction model that is directly supported by WSDL (Christensen et al. 2001) is essentially a stateless model of synchronous or uncorrelated asynchronous interactions. Although Business Process Execution Language (BPEL4WS)for Web Services (Andrews et al. 2003), can describe in which order messages have to be exchanged between services. However, flow composition still has to be specified manually whatever flow languages are used. This is not suitable for service composition in a highly dynamic Pervasive Computing environment since the number of potential compositions is large and cannot be predefined.

Another possible way is to describe the functionality of Web services with additional information, either by a semantic annotation of what it does and/or by a functional annotation of how it behaves. This kind of information should be directly processed and “understood” by a machine and therefore provides a good base for automatic service composition in a dynamic task environment. Based on Web Ontology Language (Web Ontology Working Group 2004), or OWL, an ontology for services (OWL-S)(The OWL Services Coalition 2004) is such a language. We propose to enhance the Web Services architecture with Ontologies to give an Ontology enabled Service Oriented Architecture for Pervasive Computing(Ni & Sloman 2005) and argue that it provides a promising solution to general issues, including service composition, in Pervasive Computing.

4. Service Composition

Researchers of OWL-S have developed a Service Model for Ontology that draws upon well established work in a variety of fields, including AI, planning languages, programming languages, distributed systems, process modeling and workflow technology etc. The provision of a knowledge representation of the properties and capabilities of Web services enables the automation of service composition. Disregarding grounding issues, we might be able to consider service composition as a planning and plan execution problem given the preconditions and effects of related Web services. Even so, this application domain still has many distinctive features that require and support tailoring.

As a planning task, service composition over devices may not have explicit goals, e.g., a guest in a hotel might need help to find possible functionality combination over devices. Another important feature of service composition in pervasive computing is that available information, including services themselves, world states, and objects, is partial and dynamic, e.g., a tourist in a guest room can only access devices, services, states, objects related to his room when he stays there. With classical planning, all objects are available in the initial state and the actions change the state of the objects, whereas Web services could create new objects at running time, i.e., a wireless camera startcapturing service could produce a new object videostream after its execution that could be further processed and consumed, suppose that the camera does not support broadcasting mode, by other services, like a playingvideo service running in a PDA.

Moreover, the “objects” manipulated by Web services are typed messages, which could have a very rich structure. A typed message may contain identifiable parts that can be arbitrarily complex description, e.g., a parameterType described by ontologies using OWL, which is quite different from bare objects which are used by the planning community. The rich structure of input and output messages of Web services is essential to specify the flow logic and to provide mechanism for message correlation too.

Many action descriptions in planning do not explicitly include the agent which performs the actions, but generally the client and the server of service should be given because there are too many software agents, devices, and people involved in service composition in Pervasive environment. Many services perform similar functions, so service composition must choose between several services, each sharing some of same outputs and effects. Hence, service composition is shorter but broader than typical planning problem.

We identify these issues and will try to solve some of them in this paper, and other issues will be discussed in other paper. However, before we can go into the details of how service composition work, it's necessary to formulate problems first, because fitting them into our framework requires bending the problem description somewhat.

4.1 Formulation of problems

The preceding discussion suggests that planning techniques with their extension could be used to solve service composition problems. In order to do this, the key is to find a representation of service composition problem that is expressive enough to describe a wide variety of problems, but restrictive enough to allow efficient planning algorithms to operate over it. Fortunately, Web Ontology Language, OWL, and a Semantic Markup for Web Services, OWL-S, have already provided a good base for this. In order to save space, we will use a STRIPS-like language to describe related concepts instead of verbose XML syntax.

Representation of objects. All kinds of objects in the service composition environment are represented by logic terms, named as constant symbols. For example, HDTV001 could represent a HDTV, and a VStream001 could represent a new videostream object produced by a startcapturing service. Some objects produced by services, such as VStream001, could be consumed by other services.

Representation of states. Service composers decompose a task environment into logical conditions and represent a state as a conjunction of positive literals which describe properties of objects or relationships between objects. For instance, HDTV(HDTV001) might represents HDTV001 is a HDTV object; PlayVideo(HDTV001, VStream001) might represent a state that a HDTV named HDTV001 is playing a video stream VStream001. Currently, we only consider propositional literals and ground and function free first-order literals, because they seem good enough for our problems. Objects and related propositional literals could be added and deleted according to the change of task environment either by service or by the change of time and location. In order to simply the problem, the closed-world assumption is used, meaning that any conditions that are not mentioned in a state are assumed false.

Representation of goals. A goal is a partially specified state, represented as a conjunction of positive ground literals, e.g., PlayVideo(HDTV001, VStream001). A propositional state s satisfies a goal g if s contains all the atoms in g (and possibly others). For instance, the state PlayVideo(HDTV001, VStream001) PlayAudio(Speak002, AStream002) satisfies the goal PlayVideo(HDTV001, VStream001). Note: we do not apply goal concept in problems described here directly, and the concept will be used in our work related to goal-based service composition.

Representation of services. As specified in OWL-S, the state change of a service is represented by the preconditions that must hold before it can be executed and the effects that ensure when it is executed whereas the information flow of a service is represented by input and output. In order to solve possible inconsistency between arguments in a service and state changes by the service, we define that there must be a logical literal in preconditions or effects, which captures the semantics of a parameter, for each parameter. For example, a startcapturing service by a camera is:

```
Service( StartCapturing,
  Input  cameraname: String
  Output  vstream: Videostream
  Precondition  Wirelesscamera(cameraname)
  Effect  CaptureVideo(cameraname, vstream)
          ∧ Videostream(vstream)
)
```

This is more precisely called a service schema, meaning that it represents a number of different services that can be derived by instantiating the variables camera and vstream to different constants. In general, a service schema consists of five parts.

The service name or ID, such as StartCapturing, serves to identify the service.

The input list defines the information that is, under some conditions, required for the performance of the service. Each input could have its parameter type. The output list represents process the information that the service provides to the requester. Each output could have its parameter type. The match type between input and output will be discussed in later section. Because of parameter binding during the service composition, we do not apply unique name assumption. That means two instance of different parameters could be the same even if they have different names.

Preconditions is a conjunction of function free positive literals stating what must hold in a state before the service can be invoked, including semantics of each parameter. Any variables in the precondition must also appear in the service's parameter list.

The effect is a conjunction of function free literals describing how the state changes when the service is invoked. A positive literal P in the effect is asserted to be true in the state resulting from the action, whereas a negative literal $\neg$P is asserted to be false. Variables in the effect must also appear in the service's parameter list.

Representation of participants. A service generally involves at least two participants, the client who composes and invokes the service and the server who provides the service. The client could be a software agent or a real person. For instance, the previous example of StartCapturing service schema lacks of service participants: service composer and device. Both client and server are objects, and should be instantiated before the invocation of the service. They are also could be used into the precondition and effect if required.

```
Service( StartCapturing,
  Client composer: agent
  Server camera: device
  Input  cameraname: String
  Output  vstream: Videostream
  Precondition  Wirelesscamera(cameraname)
  Effect  CaptureVideo(cameraname, vstream)
          ∧ Videostream(vstream)
)
```

Having defined the syntax for representations of service composition problems, we can now define the semantics. The most straightforward way to do this is to describe how services affect states. First, we say that a service is invocable in any state that satisfies the precondition; otherwise, the service has no effect. For a first-order service schema, establishing invocability will involve a substitution for each variable in the precondition. For example, suppose the current state is described by

```
Wirelesscamera(WC1)
```

This state satisfies the precondition

```
Wirelesscamera(cameraname)
```

with substitution {cameraname/WC1}. Thus, the concrete service StartCapturing is invocable.

Starting in object space o and state s, the result of perform an invocable service a is a new object space o' and a state s' that is the same as o and s except that any new object in the output of a is added to o' and any positive literal P in the effect of a is added to s' and any negative literal ¬P is removed from s'. Moreover, if there is a negative literal which describes the semantics of an object only, the object should be removed from new object space o'. Thus, after StartCapturing, the current object space and state becomes

```
Object: {WC1, VStream001}
State: Wirelesscamera(WC1)∧
       Videostream(VStream001)∧
       CaptureVideo(WC1, VStream001)
```

Note that if a positive effect is already in s it is not added twice, and if a negative effect is not in s, then that part of the effect is ignored. For now, we assume every literal not mentioned in the effect remains unchanged in order to avoid frame problem.

Finally, we can define the solution for a service composition problem. In its simplest form, this is just a service sequence that, when invoked in the initial state, results in a state that satisfies the terminal condition or goals.

4.2 Connection between Services

We already know that a distinct feature of Web service composition, compared to planning, is a service could “create” a new object as output and hence “create” a new predicate related to that object and/or new predicates related that object with other objects existed. For instance, a startcapturing service from a wireless camera could produce a new VStream001 object and a new propositional literal videostream(VStream001).

```
Service( StartCapturing,
  Input  WC1: String
  Output  VStream001: Videostream
  Precondition  Wirelesscamera(WC1)
  Effect   CaptureVideo(WC1, VStream001)
           ∧ Videostream(VStream001)
)
```

If our PDA provides a service PlayingVideo like the following,

```
Service( PlayingVideo,
  Input  PDA1: String,
         VStream002: Videostream
  Output  N/A
  Precondition  PDA(PDA1) ∧
          Videostream(VStream002)
  Effect  PlayVideo(PDA1, VStream002) ∧
          ¬Video(VStream002)
)
```

a possible service composition exists between Startcapturing service from a wireless camera and PlayingVideo service from a PDA. Some service inputs could be instantiated by their service providers automatically, such as WC1 from StartCapturing service and PDA1 from PlayingVideo. In order to successfully compose two services, the input which is not instantiated by its service provider and/or its service client must be able to bind with the instantiated output from another service, and they must have same parameterType or have subclass or superclass relationship if using ontologies that will be discussed in the following section in detail. After the binding, the two parameters become the same object because the latter input is actually instantiated by the former output. For example, after binding the VStream002 in PlayingVideo with VStream001 in StartCapturing, VStream002 became the same object of VStream001. Thus, Videostream(VStream002) holds because Videostream(VStream001) holds after the execution of StartCapturing service, and hence PlayingVideo is invocable now. It is worth noting that VStream001 object is consumed by PlayingVideo service. As we can see, the newly produced object in service plays a key role during the service composition, which makes sense because the main functionality of such service is to produce new objects. As a general rule, type match and parameter binding is second to none in service composition between services which produce new objects or consume new objects. Precondition of services is normally used to guarantee the semantics of related parameters and their relationships, but effect is still used to judge whether the goal is satisfied or not and to illustrate the semantics of service composition.

4.3 Exact Match and Generic Match

As discussed in previous section, the type matching between parameters is very important during service composition. The parameter descriptions in the service profile allow defining two different types of matches between services, an exact match and a generic match. An exact match is defined between two parameters which are restricted to the same OWL class or the same XML Schema type in their Service Profile descriptions. For example, the match between VStream001 and VStream002 is an exact match because they exactly belong to same OWL class videostream.

The match between the services whose output OWL Class type is a subclass of the other service's input OWL class type is called a generic match. When the output of a service is subsumed by the input, the output type can be viewed as a specialized version of the input type and these services can still be chained together. It is easily understood because all instances of a subclass are also instances of a superclass. For instance, we could define a videostream ontology for video stream because so many video type: realmedia, windowsmedia, mpeg, divx, and mpeg2 etc. A PlayingVideo service which could play all kinds of media types apparently could be composed with a CapturingVideo service which generates a windows media stream, but not vice versa.

As a rule, the services that supply an output of an exact match are more likely to be preferred in the composition and these services are displayed at the top of the matching services list. The generic matches are put at the end of the list since they are less likely to be meaningful.

5. Ad hoc service composition

Ad-hoc service composition is mainly for device control and performs serendipitous compositions of any available services to find those which may be of use to non-expert users, who will not know in advance how to achieve their goals or even what goals could be potentially available. Recall the scenario that we describe in section 2, we propose the service composition over devices could be a possible solution for problems of device control in Pervasive environment.

5.1 Device Classification

Devices can be classified into two categories: embedded devices such as a security camera which generally only provides specific functionality and does not provide a versatile input device like a keyboard and interaction devices such as a PDA which can provide many different kinds of services as well as controlling the embedded devices. Due to the progress of technology, the classification may be blurring for some advanced devices, such as a game console. A high-end game console may have a more powerful CPU and GPU than a generic PC does and could also have its own Keyboard. However it is reasonable that a low-end game console is considered an embedded device.

5.2 Classification of Services

Embedded Devices typically support both a function service and some control services. We still use services provided by the wireless camera to illustrate our idea. StartCapturing service will generate a new object Videostream VStream001. Because the service is the main functionality of the camera, in other words, what the camera is made for, we call it functional service. An advanced wireless camera could support more features, such as panning, tilting, zooming in and zooming out. Camera can expose these features as services too. However, this kind of services neither produces any new objects nor changes any world states. Instead, they are often used by person directly to control the behavior of the camera for better image. According to their usage, we call them control services.

We argue that only functional services will be involved in ad hoc service composition. Instead, control service will be exposed to interaction device directly. We will use examples in our scenario to illustrate the concept. Before that we assume devices in these examples support necessary ontology-based service description.

In essence, there is no service composition in example 1. Embedded devices such as HDTV broadcast their control services. Interaction devices can take advantage of it to control embedded devices. In OSOA, we call the software a universal remote controller. There is already a commercial PDA product named nevo from Universal Electronics which is specially designed to implement same idea basing on UPnP technology.

Now, we consider example 2. There is no question about functional service composition between a StartCapturing service from the camera and a PlayingVideo service from the PDA. What about service composition between possible control services from the camera and related services from the PDA? Actually, a service composition involving control services would become a problem if we did not distinguish functional services with control services. The service description of control services is the first barrier. How to describe a control service for an interaction device like a PDA which may have a versatile keyboard and may need to be composed with many different kinds of embedded devices? The difficult of the service description of control services in an interaction device eventually results in the difficulty in service composition. Even the problem is solved, we still have another issue. Consider example 3, we have one interaction device and three embedded devices. The dataflow of the functional service composition is PDA->Game Console->HDTV and PDA->Speaker. If we do not distinguish functional services and control services, should the data flow of possible control service composition follows the same way? Apparently, it should be PDA<->Game Console, PDA<->HDTV, and PDA<->Speaker, see Figure 1. No matter how many embedded devices are involved in an ad hoc service composition. The control flow is always from an interaction device to embedded devices and the response, such as current volume, is from embedded devices to the interaction device, which accords with our customs. If so, why don't we choose a straightforward way to expose all relative control services from embedded devices to the interaction device? There is no need to compose control services between embedded devices and the interaction device at all. This solves both difficulties at the same time. There is an assumption that parameters of control services are not so complex for embedded devices, but the assumption seems to be reasonable.

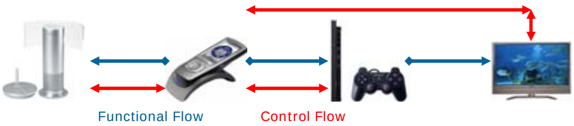

Figure 1 Functional Flow and Control Flow

Example 3 is our focus and we will illustrate our idea and algorithm basing on it. The reason is not because of four devices involved in the service composition but because of the match between multiple outputs and inputs. The PDA hosts a X3D file preprocessing service which has 2 outputs, one is unprocessed 3D data flow and one is processed Audio data flow because PDA has enough process power to produce audio stream. A game console is used to render the 3D data into video signal which is finally displayed by a HDTV.

5.2 Problem Description

The first step is the formalization of the problem described in example 3.

```
Initial Object {PDA01, GC01, HDTV01, SPK01,
                FILE01.X3D}

Initial State {PDA(PDA01),
               GameConsole(GC01),
               HDTV(HDTV01),
               Speaker(SPK01),
               X3DFile(FILE01.X3D)}

Available Service {

Service( X3DfilePreprocess
  Input: PDA01, FILE01.X3D
  Output: AStream01, 3DStream01
  Precondition: PDA(PDA01)∧
    X3DFile(FILE01.X3D)
  Effect: ¬X3DFile(FILE01.X3D) ∧
    AudioStream(AStream01)∧
    3DStream(3DStream01)
)
//Produce two new objects: AStream01 and
//  3DStrea01
//Produce two new predicates:
//  AudioStream(AStream01)∧
//  3DStream(3DStream01)
//Consume an old object and related
//predicate FILE01.X3D and
//  X3DFile(FILE01.X3D)
//Note only negate predicate that describe
//the semantic of the object can delete
//object

Service( PlayingAudio
  Input: SPK01, AStream02
  Output: None
  Precondition: Speaker(SPK01) ∧
    AudioStream(AStream02)
  Effect: ¬AudioStream(AStream02) ∧
    PlayAudio(SPK01, AStream02)
)
//Consume an old object and related
//predicate AStream02 and
//AudioStream(AStream02)

Service( 3DRendering
  Input: GC01, 3DStream02
  Output: VStream01
  Precondition: GameConsole(GC01)∧
    3DStream(3DStream02)
  Effect: ¬3DStream(3DStream02) ∧
    VideoStream(VStream01)
)
//Produce one new objects: VStream01 and
//related predicate: VideoStream(VStream01)
//Consume an old object and related
//predicate 3DStream02 and
//3DStream(3DStream02)

Service( PlayingVideo
  Input: HDTV01, VStream02
  Output: None
  Precondition: HDTV(HDTV01) ∧
    VideoStream(VStream02)
  Effect: ¬VideoStream(VStream02) ∧
    PlayVideo(HDTV01,VStream02)
)
//Consume an old object and related
//predicate VStream02 and
//VideoStream(VStream02)

}
```

As mentioned in previous part, ad hoc service composition is used to help non-expert users to control devices which they are even not familiar with. Hence there is no explicit goal. However, because the heart in ad hoc service composition is producing new object and consume old object, a service without an output as new object could act as the final service to end the composition. We start from the task environment's initial object and initial state, considering the sequence of services until we find a sequence that reaches a service without an output. In this situation forward object-space and state-space search is the obvious choice for service composition on devices.

There are two sets in the beginning of the search, initial set of objects and initial set of world state. In general, each state will be a set of positive ground and function free first order literals, aka predicates, and literals not appearing are false.

The services that are applicable to a state are all whose inputs could be instantiated and all whose preconditions are satisfied. The successor state resulting from a service is generated by adding the new objects and deleting consumed objects, adding the positive effect literals and deleting the negative effect literals. We use the previous example to illustrate the details.

Step One:

```
object set { PDA01, GC01, HDTV01, SPK01,
  FILE01.X3D }
state set {PDA(PDA01), GameConsole(GC01),
  HDTV(HDTV01), Speaker(SPK01),
  X3DFile(FILE01.X3D)}
```

Check the available service, we find only X3DfilePreprocess service all whose input could be instantiated and all whose precondition are satisfied. After the execution of the service, new objects(AStream01, 3DStream01) and related predicates(AudioStream(AStream01)∧3DStream(3DStream01)) are produced and an old object(FILE01.X3D) and related predicate(X3DFile(FILE01.X3D)) are consumed.

Step Two:

```
object set { PDA01, GC01, HDTV01, SPK01,
  AStream01, 3DStream01}
state set {PDA(PDA01), GameConsole(GC01),
  HDTV(HDTV01), Speaker(SPK01),
  AudioStream(AStram01),
  3DStream(3DStream01)}
```

Now we can find two services are applicable: PlayingAudio and 3DRendering. Repeat the same procedure in Step One. If using a breath first search, a service without a new object as its output is the leaf of search space, e.g., PlayingAudio service. All paths from the service node after initial object and state set node to the leaf service node in the search tree represent all executable service composition in the task environment, see Figure 2. Finally, we get two possible service compositions from the problem description.

```
X3DFilePreprocess->PlayingAudio
X3DFilePreprocess->3DRendering->
  PlayingVideo
```

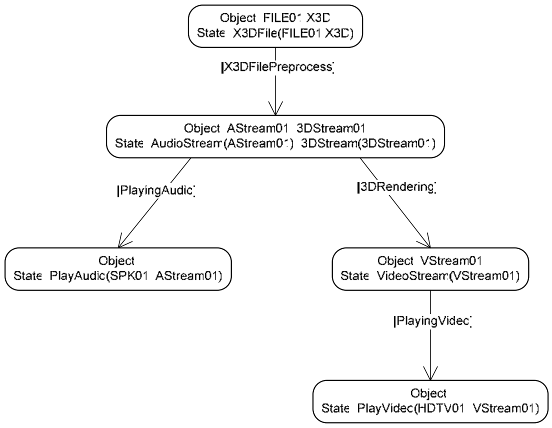

Figure 2 Simplified composition flow

Recall that, in the absence of functional symbols in preconditions and effects, the object and state space of a service composition problem is limited. Therefore, any search algorithm that is complete will be a complete service composition algorithm. So called completeness means all terminated paths could be found. Consider the task environment, many services (breadth), few steps (depth), and no obvious goal; we believe that a depth-limited (normally 4) search might be a reasonable choice because most feasible service composition on devices only have a depth of 2.

The state literals themselves can help explain the meaning of possible service composition, such as, the final state after these two service composition in the previous example is a set {PlayAudio(SPK01, AStream01) ∧ PlayVideo(HDTV01, VStream01)}, and the forward space search could even show the composition procedure to users step by step. It makes sense because abstract service parameter type makes little help to assist user to understand the "meaning" of service composition.

We are aware of disadvantages of forward space search. The first one is the irrelevant service problem. Actually, the problem is a strength not a weak point in ad hoc service composition on devices. Because our intention is help non-expert user to find all possible function composition over devices, it is reasonable that all applicable services are considered. Second, the approach quickly bogs down without a good heuristic. This is not an issue for our intention too. Consider the deploy environment, a hotel room, a house, and a conference room etc, available devices (services) are usually no more than 1000. More importantly, the possible service compositions over devices normally are no more than 4 steps, which is not like typical planning problems and other kind of service composition problems e.g. e-business. Classification through service profile hierarchy and user's preference and constraints could help reduce service space further.

It is worth noting that no service is really executed during the service composition. The instantiating and binding parameter and producing and consuming object are just simulation. Only after a user chooses his favorite service composition and executes the composition, these services are really invoked, see Figure 3.

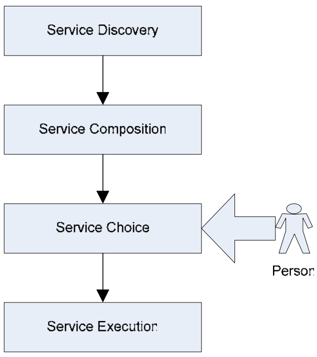

Figure 3 Ad hoc service composition flow

6. Related Work

Task Computing(Masuoka et al. 2003;Masuoka, Parsia, & Labrou 2003) only deals with type match and limits that service can only have one input/output in order to implement service composition.

Work in (Mcllraith & Son 2002) needs to predefine plan templates, and does not consider the importance of type match. (Wu et al. 2003) has similar issues.

Semantic Graph based Service Composition (Fujii & Suda 2004) seems to has some deficits in their match rules.

(Bellur & Narendra 2005) is an ongoing work, but at least they need to predefine workflows.

7. Conclusion

We present an approach, Ontology-enabled Service Oriented Architecture, for general issues in Pervasive Computing and identify automatic service composition as the core part in OSOA. Basing on the semantic description of Web services, we argue planning could be applied in service composition but need to be customized somewhat. We analyze the difference between planning and service composition and present our approach to realize ad hoc service composition.

8. References

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., & Klein, J. 2003, *Business Process Execution Language for Web Services Version 1.1*.

Bellur, U. & Narendra, N. C. "Towards Service Orientation in Pervasive Computing Systems", in *International Conference on Information Technology 2005 (ITCC2005)*, pp. 289-295.

Christensen, E., Curbera, F., Meredith, G., & Weerawarana, S. 2001, *Web Services Description Language (WSDL) 1.1*.

Fujii, K. & Suda, T. "Dynamic Service Composition Using Semantic Information", in *the 2nd International Conference on Service Oriented Computing (ICSOC' 04)*.

Masuoka, R., Labrou, Y., Parsia, B., & Sirin, E. 2003, "Ontology-Enabled Pervasive Computing Applications", *IEEE INTELLIGENT SYSTEMS*, vol. 18, no. 5, pp. 68-72.

Masuoka, R., Parsia, B., & Labrou, Y. 2003, "Task Computing - the Semantic Web meets Pervasive Computing", in *The 2nd International Semantic Web Conference 2003 (ISWC 2003)*.

Mcllraith, S. & Son, T. C. "Adapting Golog for Composition of Semantic Web Services", in *International Conference on the Principles of Knowledge Representation and Reasoning (KRR'02)*, pp. 482-496.

Ni, Q. & Sloman, M. "An Ontology-enabled Service Oriented Architecture for Pervasive Computing", in *Internatiola Conference on Information Technology 2005*, pp. 797-798.

The OWL Services Coalition 2004, *OWL-S: Semantic Markup for Web Services*.

Web Ontology Working Group 2004, *Web Ontology Language (OWL)*.

Wu, D., Parsia, B., Sirin, E., Hendler, J., & Nau, D. "Automating DAML-S Web Services Composition Using SHOP2", in *Proceedings of the Second International Semantic Web Conference (ISWC2003)*.