Composition of Services in Pervasive Environments: A Divide and Conquer Approach

Gilbert Cassar, Payam Barnaghi, Wei Wang, Suparna De, Klaus Moessner
Centre for Communication Systems Research
University of Surrey,
Guildford, UK, GU2 7XH
{*g.cassar, p.barnaghi, wei.wang, s.de, k.moessner*} *@surrey.ac.uk*

***Abstract*—In pervasive environments, availability and reliability of a service cannot always be guaranteed. In such environments, automatic and dynamic mechanisms are required to compose services or compensate for a service that becomes unavailable during the runtime. Most of the existing works on services composition do not provide sufficient support for automatic service provisioning in pervasive environments. We propose a *Divide and Conquer* algorithm that can be used at the service runtime to repeatedly divide a service composition request into several simpler sub-requests. The algorithm repeats until for each sub-request we find at least one atomic service that meets the requirements of that sub-request. The identified atomic services can then be used to create a composite service. We discuss the technical details of our approach and show evaluation results based on a set of composite service requests. The results show that our proposed method performs effectively in decomposing a composite service requests to a number of sub-requests and finding and matching service components that can fulfill the service composition request.**

***Keywords*-Service Composition; Sensors; Semantics;**

I. Introduction

Service composition is an essential task in service oriented computing to build complex business systems and applications from large number of potentially simple, distributed and heterogeneous services. Most of the existing methods for service oriented computing were primarily developed for the carefully designed and maintained Web services to build sophisticated enterprise systems and applications. In pervasive environments such as the Internet of Things (IoT) where services are often dynamic, mobile, less reliable and device-dependent (e.g., a sensor service is dependent on the status of the sensor which is often resource-constrained), those existing methods face significant challenges and need to be adapted. The first step in addressing this challenging problem is to design a compatible while simpler description model for real world services operating in pervasive environments. Methods for service composition as well as discovery in IoT need to be more efficient than those for general Web services (due to the number of real world services); effective compensation mechanisms are also needed to ensure the continuity of composed service at runtime when service components inside the composite one become unavailable (due to the dynamic and unreliable nature of the pervasive environments such as device mobility or network disrruption).

In this paper we present our recent research on semantic service composition in pervasive environments such as the IoT. The main contribution is the design of the *Divide and Conquer* method for service composition and compensation based on iterative decomposition of service requests and matching against service descriptions. We demonstrate a novel concept called *Transient Link Dependency Matrix* which facilitates the construction of the service composition solution. Based on the concept, we explain how a (relatively complex) service request can be broken into one or more simpler sub-requests which represent relaxation of the original search criteria and make the process of searching and matching more flexible. Furthermore, we show that the proposed Divide and Conquer approach can also be used to compensate services during service execution time. The rest of the paper is organised as follows. Section II and III describe the semantic service description model and service search and matchmaking methods, respectively, which are the foundations for the service composition method. Section IV introduces the concept of Transient Link Dependency Matrix and elaborates how it can be constructed with the splitting of service requests to facilitate matching with service descriptions. Section V explains in detail our Divide and Conquer algorithm for service composition as well as compensation based on the Transient Link Dependency Matrix. Section VI briefly describes how service execution plans can be automatically constructed using the proposed algorithm. Section VII demonstrates the experimental results and performs a comparison study with the state-of-the-art. Finally Section VIII concludes the paper and outlines future research work.

II. Service Representation

The service description model that forms the basis of the service composition algorithm proposed in this paper is part of our earlier work detailed in [1]. Here, we briefly present some of the important concepts and properties of the model that are pertinent to service composition.

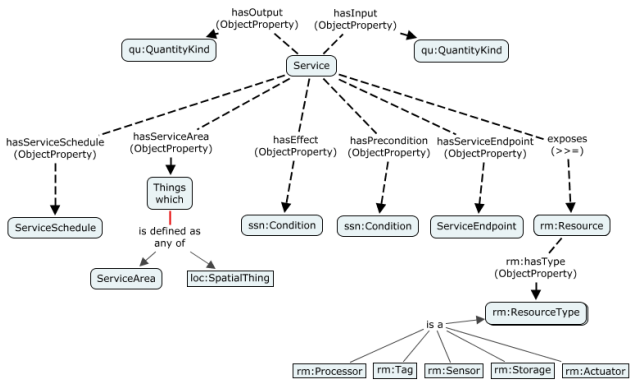

Figure 1. Service Model.

Due to the different types of connected devices possible in pervasive environments, and the hardware and software heterogeneity, the service descriptions have been modelled to provide a uniform abstraction for exposing the functionalities provided by those devices (see Figure 1).

The service is mapped to the underlying device through the "exposes" property to a resource, which has different types (*rm:hasType property*) depending upon the type of the connected device. The resource abstraction provides both hardware (*e.g.* sensor, actuator) and software specification (*e.g.* platform of a storage component) of the connected device. Since the service exposes the underlying resource functionalities, the resource attribute that is exposed through a service either as output data type (hasOutput) or as an input parameter (hasInput) is captured in the service description. The input and output parameters are specified in terms of the generic instance quantities from the Ontology for Quantity Kinds and Units (QU)[1], such as *temperature* or *luminosity*. For actuator services, the impact on the real-world attribute after service execution is modelled through the *hasEffect* parameter. Similarly, any pre-conditions that need to be met before service execution can be specified through the *hasPrecondition* parameter. The actual technology used to invoke the service is modelled through the *hasServiceType* parameter, which could take a value such as *REST* for a RESTful Web Service. The operation area of a service, represented through the *hasServiceArea* property, can be specified in terms of polygons indicating map areas (*e.g.* a rectangular area with the diagonal latitude, longitude coordinates defined) or in terms of concepts from an indoor location model or location ontologies such as GeoNames[2]. The *hasServiceSchedule* property allows specifying time constraints on service availability. The service also has ID (*hasID*) and name (*hasName*) properties. An OWL-DL representation of the service description and an example temperature sensor service are available at http://ccsriottb3.ee.surrey.ac.uk:8080/IotaDataFiles/models/Service.owl and http://ccsriottb3.ee.surrey.ac.uk:8080/IotaDataFiles/models/ServiceInstance.owl, respectively.

[1]http://www.w3.org/2005/Incubator/ssn/ssnx/qu/qu-rec20.html#Section_dim

[2]http://www.geonames.org/ontology/documentation.html

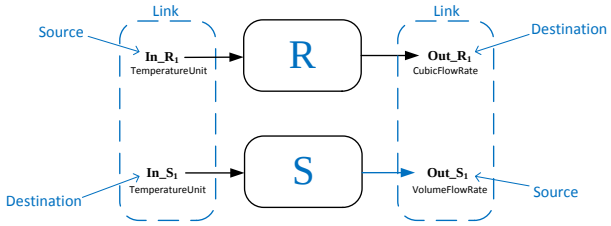

Figure 2. Source and destination parameters in service links.

We refer to a service description as a tuple $S = \{in, out, proc\}$. The sets $in(S)$, $outS$, and $proc(S)$ denote respectively the sets of inputs, outputs, and processes of service S. Similarly, we consider a request R to be a template written in the same format as the semantic model defining S. The request R defines the desired inputs and outputs using semantic definitions while also defining any internal processes and operations required by the service. However, the proposed solution is not limited to only one service modelling technology and any model that can be expressed as a tuple $S = \{in, out, proc\}$ can be used. In the following section, we outline the method for service search and matchmaking (based on the semantic service representations) which is essential for dynamic service composition.

III. Service Search and Matchmaking

Dynamic service composition approaches are highly dependent on the efficiency of the underlying service search and machmaking mechanism that they use. The search and matchmaking mechanism is responsible for finding the group of services from which the candidate services for composition are chosen. The more relevant the services found by the service search and matchmaking, the more efficient the service composition approach will be. The work in this paper builds upon our previous work on service search, matchmaking, and ranking [2].

Apart from matching a service to a request/sub-request, service composition also requires a method for measuring the compatability of individual service *IO* parameters to the *IO* parameters of a request. Logical signature matching has been used in different works to verify whether the *IO* parameters of a service are compatible with the *IO* parameters of another service [3], [4]. Lécué *et. al.* [4] propose a composition algorithm which finds semantic compatabilities among independently defined service descriptions using the semantic matchmaking between the input and output service parameters, which they refer to as *Causal Links*.

In [2] we define a *link* as a logical relationship between two *IO* parameters. A link has a source parameter $Source$

and a destination parameter $Destination$ (as shown in Figure 2) and is denoted as $Link(Source, Destination)$. Links in automated service matchmaking can represent a possible connection between two services, the relevancy of an input of a service to one of the input parameters specified in a service request, or the ability of a service to generate one of the outputs specified in a service request.

Let τ be a domain ontology model. Let $Source$ be a source IO parameter concept and let $Destination$ be an IO parameter concept that $Source$ can be linked to. Then, the type of link between $Source$ and $Destination$: $Link(Source, Destination)$ can be classed as one of the four categories explained below:

1) **Exact**: $Source$ is an exact match to $Destination$ if $\tau \models Source \equiv Destination$.
2) **PlugIn**: $Source$ plugs into $Destination$ if $\tau \models Source \sqsubseteq Destination$.
3) **Subsumes**: $Source$ subsumes $Destination$ if $\tau \models Source \sqsupseteq Destination$.
4) **Disjoint**: $Source$ is not related to $Destination$ in any of the above ways.

Individual link analysis makes it possible to dissect the degree of match between a service and request and enables a fine grained matchmaking compared to *IO* matchmaking filters. This approach stems from the concept that the most important part in a service request is the outputs and as long as all the required outputs can be provided by a service, it does not matter if the service can produce extra outputs that will not be used. Similarly, if a request specifies that the client is capable of suplying certain parameters as inputs, it does not matter if the targeted service only requires a subset of these available inputs to work. Thus we propose a matchmaking mechanism that works by assigning weights to individual links. The degree of match between a service and a request is then given by summing together the weights of the individual links.

In this work, we use the hybrid matchmaking method described in [2] to search for the list of services that match to a request/sub-request during composition. They hybrid matchmaking method combines the probabilistic matchmaking method described in [5] to a link-based matchmaking [2] described above thus increasing the accuracy of the results. The list of search results returned by the matchmaking method ranks the most relevant services at the top of the list. The ranking makes it easier to pick the candidate services that match to a request/sub-request.

IV. Transient Link Dependency Matrix

While automatically building a composite service, it is important to keep a model of the structure being built and considering how the input parameters of a service may depend on the output parameters of another service. Omer and Schill [6] use an Input/Output *Dependency Matrix* to represent how the inputs of candidate services chosen for composition depend on the outputs of other services. They propose a method to automatically generate an execution plan for the composite web services based on the constructed Dependency Matrix. However, their approach allows for cyclic dependencies [6]. A cyclic dependency occurs when the output of a service is matched to the input of a service that generates an output that feeds back into the first service, thus creating a loop. We argue that a service can only generate its outputs if it is given the required inputs first. This implies that if a web service A depends on the web service B and B depends on one of the outputs of A, the outputs of A will not be generated until B generates its outputs and B cannot generate its outputs unless it receives all the required inputs. The latter will create an intractable loop. Thus cyclic dependencies should not be allowed in a service composition.

We propose a *Transient Link Dependency Matrix* (TLDM), similar to the Dependency Matrix proposed by Omer and Schill [6]. However, instead of having an nxn matrix where n is the number of candidate service for composition, we propose an sxd matrix where s is the total number of destination parameters that need to be matched and d is the total number of available source parameters. Inputs specified in a service request are parameters that can be supplied by the user and thus take the role of source parameters. Outputs specified in the service request are parameters that must be generated by the composite service and take the role of destination paramters in the TLDM. For candidate services chosen for composition, the input parameters take the role of destination parameters that need to be matched to source parameters, and the outputs take the role of source parameters that can be fed into destination parameters. Each column can contain no more than one entry with an x that indicates which source parameter that destination parameter is matched to. Source parameters can be matched to more than one destination parameter. Empty columns indicate destination parameters that have not been matched to any source parameters yet. This set up makes it easier to represent exactly how the interface of one service links to the interface of other services. If one of the columns representing inputs of a service is empty, it means that all the necessary information cannot be supplied to that service and the output of that service cannot yet be used in the composition. In our approach, we make sure no cyclic dependencies are created by making the outputs of a service available for link matching only after the inputs for that service are completely matched.

Another main difference between the TLDM and the Dependency Matrix proposed by Omer and Schill [6] is that the TLDM starts as two sub-matrices: the left hand side and the right hand side matrices. Source parameters listed in the left hand side matrix are also available in the right hand side but the right hand side matrix also contains probational source parameters that are not available on the left hand

Algorithm 1 Divide and Conquer Approach

Require: TheRequest
1: $candidateList \leftarrow StrictServiceDiscovery(TheRequest)$
2: **if** $candidateList$ is NOT EMPTY **then**
3: Solution Found
4: **else**
5: $LoadTheRequestinTLDM$
6: $RHSplit \leftarrow TheRequest.outputs$
7: $candidateList \leftarrow ServiceDiscovery(RHSplit)$
8: **if** $candidateList$ IS EMPTY **then**
9: No Solution Exists
10: **else**
11: $matchIO()$
12: $LHSplit \leftarrow TheRequest.inputs$
13: $candidateList \leftarrow ServiceDiscovery(LHSplit)$
14: **if** $candidateList$ is NOT EMPTY **then**
15: $matchIO()$
16: **end if**
17: $int\ depth \leftarrow 2$
18: **while** depth $<$ treshold AND TLDM.solutionNotFound **do**
19: $NewRequest.inputs \leftarrow TLDM.useableOutputs$
20: $NewRequest.outputs \leftarrow TLDM.unmatchedInputs$
21: $candidateList \leftarrow ServiceDiscovery(TheRequest)$
22: **if** $candidateList$ is EMPTY **then**
23: $RHSplit \leftarrow NewRequest.outputs$
24: $candidateList \leftarrow ServiceDiscovery(RHSplit)$
25: **if** $candidateList$ is NOT EMPTY **then**
26: $matchIO()$
27: **end if**
28: $LHSplit \leftarrow NewRequest.inputs$
29: $candidateList \leftarrow ServiceDiscovery(LHSplit)$
30: **if** $candidateList$ is NOT EMPTY **then**
31: $matchIO()$
32: **end if**
33: **else**
34: $matchIO()$
35: **end if**
36: depth++
37: **end while**
38:
39: **if** TLDM.solutionFound **then**
40: Solution Found
41: **end if**
42: **end if**
43: **end if**

side. The two sub-matrices eventually converge to become a single matrix, at this point the probational source parameters stop being probational and the service composition solution is complete The left hand side matrix starts building on top of inputs specified in the service request and the sensors (which do not require any inputs to produce their output). The right hand side matrix starts from the required outputs and builds its way back, therefore the inputs of these services have not yet been properly matched and thus their outputs are only available on a probation status. When all the inputs of a service on the right hand side of the matrix get matched to outputs from the left hand side, the outputs of that service are transfered to the left hand side matrix and are not on probation state anymore. When all the required outputs are linked to outputs from the left hand side, it means the service composition solution is complete. The process of populating the TLDM is explained in more detail in the next section.

V. Divide and Conquer Approach

The Divide and Conquer approach for service composition is an iterative algorithm designed to split a request into

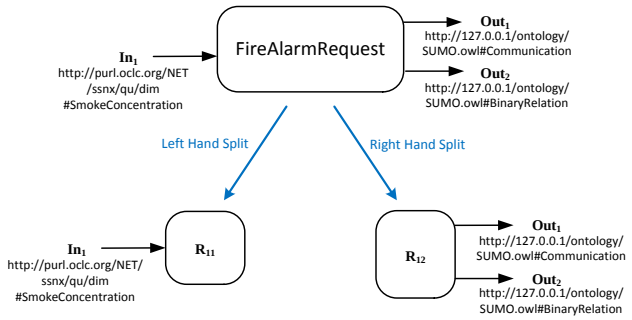

Figure 3. Splitting request *FireAlarmRequest* into sub-requests $R_{1,1}$ and $R_{1,2}$.

simpler sub-requests that relax the matchmaking criteria and make it easier to find relevant candidate services. The method also actively checks at every step of the process how the retreived candidate services can be used together for composition while inherently avoiding the creation of cyclic dependencies among the candidate services.

Algorithm 1 shows the pseudo code for the Divide and Conquer approach. The process starts with a request that specifies a set of input and output parameters and also a set of semantic descriptions that specify the required internal processes and operations. Formally, a request R can be expressed as $R = \{in, out, proc\}$ where the sets $in(R)$, $outR$, and $procR$ denote the sets of required inputs ,outputs, and processes respectively. If searching directly for R yields no candidate service that can individually provide all the required capabilities, we split up the request R into two sub-requests as shown in Figure 3. The left hand split $R_{1,1}$ creates a search in terms of inputs and the right hand split $R_{1,2}$ creates a search request in terms of outputs.

The algorithm takes $in(R)$ and $proc(R)$ to create sub-request $R_{1,1}$ and $out(R)$ and $proc(R)$ to create $R_{1,2}$ (as shown in Figure 3). Note that when we split a request, we use the convention $R_{a,b}$ where a indicates how many internal levels we have split the original request, and b takes values $\{0, 1, 2\}$. $b = 0$ denotes a request which has not been split, $b = 1$ denotes the left hand split, and $b = 2$ denotes the right hand split.

The TLDM is populated by searching for services that can satisfy $R_{1,2}$. If no service is found that contains all the output parameters specified in $in(R_{1,2})$, the first set of services $S_{1,2}$ that collectively contain all the output parameters specified in $out(R_{1,2})$ are selected. *i.e;*

$$\forall\ S\_Out_i \in out\{S_{1,2}\}\ \exists\ R_{1,2}\_Out_j \in out\,(R_{1,2})\ :\ Link(S\_Out_i,\ R_{1,2}\_Out_j) \in \{Exact, Subsumed\}$$

For every output $S\_Out \in in(S_{1,2})$ the algorithm checks if there exists a parameter in the destination parameters listed in the right hand side of the TLDM that it can be linked to.

If a valid match is found, S_Out_i is added as a probational source candidate in the TLDM and the link between S_Out_i and the destination parameter is stored in the matrix. This procedure is carried out by the $matchIO()$ function and is repeated until every output parameter specified in $R_{1,2}$ has been matched to a source parameter.

The search and matchmaking mechanism is called upon again to search for services that match $R_{1,1}$. The algorithm selects the first set of service $S_{1,2}$ that collectively contain all the input parameters specified in $in(R_{1,1})$. *i.e.*

$$\forall \; S\_In_i \in in\{S_{1,1}\} \; \exists \; R_{1,1}\_In_j \in in(R_{1,1}) : Link(S\_In_i, R_{1,1}\_In_j) \in \{Exact, Plug-In\}$$

For every input $S\_In_i \in in(S_{1,1})$ the algorithm checks if there exists a source parameter listed in the left hand side of the TLDM that it can be linked to. If not all the inputs of a service can be linked to source parameters in the left hand side of the TLDM, the service is discarded because without supplying all its the necessary inputs, the service cannot be composed. If a valid match is found, every S_In_i is added as a destination parameter in the TLDM and the link between S_In_i and its source parameter is stored in the matrix. The output parameters of that service are then made available as source parameters in the left hand side of the TLDM. This procedure is carried out by the $matchIO()$ function.

At this stage, the entries in the TLDM are checked to see if any source parameters from the left hand side can be linked to destination parameters from the right hand side. If any services from the right hand side get all their inputs matched to the left hand side, the service stops being on probation status and all its source parameters are transfered to left hand side. If all the services in the right hand side have had their inputs matched to the left hand side, then the algorithm has found all the candidate services necessary to create a composite service. Otherwise, a new sub-request $R_{2,0} = \{in, out, proc\}$ is created. Where $in(R_{2,0})$ are all the available source parameters listed in the left hand side of the TLDM, $out(R_{2,0})$ are all the unmatched destination parameters listed in the TLDM, and $proc(R_{2,0}) = proc(R_{1,0})$.

If a service that satisfies sub-request $R_{2,0}$ is not found, the sub-request would be split again into two parts: the left hand split $R_{2,1}$ and the right hand split $R_{2,2}$ and the process will be repeated all over again. The method is designed to loop until the inputs to all the services in the TLDM are linked. However, not all outputs need to be used in a composite service and thus outputs that have not been linked can be disregarded in the end. To ensure a fully operational composite service, it is important that the services from the right hand matrix have all their inputs matched (so that they can function and produce the required outputs). It is not necessary for all the outputs to be matched because there might be cases where a service produces some other outputs besides the outputs that we need. In our work we assume that the user does not have any problem if extra outputs are produced as long as the required outputs are ultimately generated.

The algorithm may reach to a point where no solution that eventually bridges the left hand split and the right hand split together is found and the request keeps being broken down without converging to a viable composition. This shortcoming can be overcome by defining a treshold for the maximum depth (number of splits) that can be reached by the algorithm.

The maximum depth for the divide and conquer method can be determined by a combination of criteria defined explicitly by the user or determined implicitly from the user preferences. Such criteria includes the maximum required response time (in case of real-time services), maximum amount of complexity that can be handled by the host of the composite service, and the maximum treshold of complexity desired by the user.

Another possible solution is to stop splitting, and to restart the steps but be more relaxed on the input criteria i.e. assume that the user might be able to supply additional inputs if it is needed.

VI. Automatic Creation of Exectuion Plan

Works on automatic execution plan creation for service composition such as [6] use the dependencies of services to automatically build an execution plan from a list of candidate services. As with most works in this field, the research interest is in the conceptual logic of execution path rather than generating a practical implementation [7], *e.g.* a WS-BPEL workflow[3].

In our method, the service execution plan creation is performed automatically through the TLDM while we also look for the candidate services. This is more efficient than breaking the automatic service composition into separate service discovery and automatic execution plan creation stages. Building the execution plan through the TLDM also makes sure that the final execution plan contains no cyclic dependencies (as explained in Section IV).

VII. Evaluation

The dataset used for our evaluations consists of 1220 service descriptions that include sensors, actuators, and processing services. The service descriptions are represented in the semantic service description model described in Section II. The services in the dataset contain concepts and parameters from different domains including: business, medical, city, commerce, geography, military, office, technology, travel, and weather (the dataset can be accessed at http://tinyurl.com/9a2mmlf).

We evaluate the divide and conquer algorithm against a Backward Chaining algorithm [8], an algorithm used

[3]http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

Table I
REQUESTS AND THE SERVICES THAT MAKE UP THE SOLUTION FOR EACH REQUEST.

Request Name	Request Number	Services	No. of Services
BusRequest2	#1	BusLookUp_WS, BusStopLocation_WS	2
CameraPrintRequest	#2	PictureCamera_01BA02, PhotoPrinter_BA02	2
WeatherForecastRequest	#3	Forecast_WS1, Forecast_WS2, CheckRain_WS	3
BusRequest1	#4	GPS_WS, BusLookUp_WS, BusStopLocation_WS	3
FireAlarmRequest	#5	temperature_sensor_01_BA_02, smoke_sensor_01_BA_02, fire_processing_service, fire_alarm_service	4
CityRequest1	#6	CityTemperature_WS, Rainfall_WS, Windspeed_WS, Inference_WS, Actuator_WS	5
CityRequest2	#7	Traffic_WS, CityTemperature_WS, Rainfall_WS, Windspeed_WS, Inference_WS, Actuator_WS	6

Table II
SERVICES THAT WENT MISSING AND THE SERVICES THAT MAKE UP THE COMPENSATION SOLUTION.

Service Name	Services in Compensation Solution
CameraPrintRequest	VideoCamera_01BA02, MotionPictureToStill_Service
Inference_WS	BUInference_WS1, BUInference_WS2, BUInference_WS3

for top-down service composition. The Backward Chaining algorithm works by finding services that match the outputs of the request and then takes the inputs of these candidate services and searches for other services whose outputs match the inputs of these services. The algorithm keeps working backwards like this until sensor services (that require no inputs) are found, or the inputs of the services at the bottom of the chain match the inputs specified in the service request.

The evaluation test cases consisted of submitting a service request to both algorithms and letting the methods find a composition that provides a solution for the request. We made sure that for every request there exists at least one possible composition solution in the dataset. All the requests are provided in the form the semantic service description model described in Section II. Table I shows the requests and the services that make up the solution for each request (the service requests used in our evaluations can also be found at http://tinyurl.com/9a2mmlf).

We demonstrate how the divide and conquer approach can be used to provide automated service compensation by removing service *PictureCamera_01BA02* from soloution #3 and *Inference_WS* from solution #5. The service description of PictureCamera_01BA02 and Inference_WS are submitted to both algorithms as requests and the algorithms look for services or composition of services as replacements. The services that when composed together can replace *PictureCamera_01BA02* and *Inference_WS* are shown in Table II.

All experiments were carried out on a computer with Intel(R) Core(TM)2 Duo T7500 2.2GHz CPU, 4GB RAM, and running Microsoft Windows 7 x86.

A. Results

Table III summarises the results from the service composition evaluations. For each request, we measured the time it takes for the algorithm to find the solution and the depth (*i.e.*; the number of times the request had to be split). If a table entry contains the symbol '/', it means that the algorithm did not manage to find a composition solution for that request.

Table III
SERVICE COMPOSITION RESULTS

Request No.	Divide and Conquer		Back Chaining	
	Time / s	Depth	Time / s	Depth
#1	12.629	2	11.421	2
#2	13.196	2	11.660	2
#3	14.141	2	/	/
#4	14.622	3	12.527	3
#5	14.793	2	13.279	2
#6	45.348	4	/	/
#7	/	/	/	/

Table IV
SERVICE COMPENSATION RESULTS

Compensation Request	Divide and Conquer		Back Chaining	
	Time / s	Depth	Time / s	Depth
PictureCamera_01BA02	13.532	2	12.630	2
Inference_WS	13.357	1	/	/

Table IV summarises the results from the service compensation evaluations. The service descriptions of the services that were removed from the dataset were used as the compensation request. The table shows how long each algorithm took to find a composition of services that can replace the missing service and the depth reached in each case. If a table entry contains the symbol '/' it means that the algorithm did not manage to find a solution for that compensation request.

B. Discussion

The results show that the Back Chaining algorithm converges to a solution faster than the Divide and Conquer algorithm for solutions consisting of a small number of services. This is because the Back Chaining algorithm does not need to split the request and call onto the service matchmaking mechanism twice. However the Back Chaining algorithm was not able to solve requests #3, #6, and #7 while the Divide and Conquer algorithm successfully solved requests #3 and #6.

Request #7 was similar to request #6 but specified no inputs. We included this request to show how the Divide and Conquer algorithm responds in the same way as the Back Chaining algorithm when no inputs are specified in the request to create a left hand split from. If a request is too vague, it is harder for an automated service composition approach to find the correct solution. It is important that service requests provide enough information about the

required functionality. In the case of request #7, the solution required an extra service (*Traffic_WS*) to provide information about the traffic density. The traffic density information was specified as an input in request #6 and therefore *Traffic_WS* was not required as part of the solution because the request specified that the output provided by *Traffic_WS* was already known.

The service compensation results show that the Divide and Conquer approach can find service compositions to replace missing services even in cases where the Back Chaining algorithm fails. The Divide and Conquer approach works best when the request indicates what inputs are available in conjunction with what outputs are required. If a request only specifies the required outputs, the Divide and Conquer approach can only work from the top-down and thus responds in a similar way as a Back Chaining algorithm.

The time complexity of the Back Chaining algorithm is $O\left(b^{d-1}\right)$ while the time complexity of the Divide and Conquer algorithm is $O\left(b^{\frac{d-1}{2}}\right)$. Parameter b is the *branching factor* which depends on the maximum number of input links every service can have and parameter d is the *depth* of the solution.

VIII. Conclusions and Future Work

Automatic service composition offers a solution for providing the required functionality to users in the service-oriented platforms when no single resource can provide the full required functionality. The Divide and Conquer approach revolves around the concept of representing resources as semantic services so that service-oriented computing methods can be applied to provide higher-level solutions such as service composition and service compensation. Our proposed approach provides a mechanism for breaking down a service request into simpler sub-requests. The sub-requests relax the search criteria and make the process of searching candidate services for composition more flexible. Evaluation results show that the approach managed to find the correct solution for almost all composition requests in our experiments, even in some cases where a Back Chaining algorithm failed. The Divide and Conquer approach can also be used to find replacement services for the missing ones during runtime (*i.e.*; service compensation). The evaluation results also show that the divide and conquer approach was able to find compensation solutions successfully and outperformed the Back Chaining algorithm.

Our future work will focus on automated alteration and adaptation of services based on service quality and context information to predict service quality parameters at design-time and runtime. We will also focus on mapping the execution plan that is automatically created by the Divide and Conquer approach (discussed in Section VI) from a logical workflow to a practical implementation expressed in WS-BPEL.

Acknowledgment

This paper describes work undertaken in the context of the IoT-A project, IoT-A: Internet of Things - Architecture (http://www.iot-a.eu/public) contract number: 257521. The second and third authors are also funded by the ICT IoT.est project (www.Ict-Iot.est.eu) contract number: 288385.

References

[1] S. De, T. Elsaleh, P. M. Barnaghi, and S. Meissner, "An internet of things platform for real-world and digital objects." *Scalable Computing: Practice and Experience*, vol. 13, no. 1, 2012.

[2] G. Cassar, P. Barnaghi, W. W., and K. Moessner, "A hybrid semantic matchmaker for iot services," in *Internet of Things (iThings), 2012 IEEE International Conference on*, nov. 2012.

[3] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara, "Semantic matching of web services capabilities," in *Proceedings of the First International Semantic Web Conference on The Semantic Web*, ser. ISWC '02. London, UK, UK: Springer-Verlag, 2002, pp. 333–347.

[4] F. Lécué, E. M. Goncalves da Silva, and L. Ferreira Pires, "A framework for dynamic web services composition," in *2nd ECOWS Workshop on Emerging Web Services Technology (WEWST07), Halle, Germany*. Germany: CEUR Workshop Proceedings, November 2007.

[5] G. Cassar, P. Barnaghi, and K. Moessner, "A probabilistic latent factor approach to service ranking," in *Intelligent Computer Communication and Processing (ICCP), 2011 IEEE International Conference on*, aug. 2011, pp. 103 –109.

[6] A. M. Omer and A. Schill, "Web service composition using input/output dependency matrix," in *Proceedings of the 3rd workshop on Agent-oriented software engineering challenges for ubiquitous and pervasive computing*, ser. AUPC 09. New York, NY, USA: ACM, 2009, pp. 21–26.

[7] Y. Syu, S.-P. Ma, J.-Y. Kuo, and Y.-Y. FanJiang, "A survey on automated service composition methods and related techniques," in *Services Computing (SCC), 2012 IEEE Ninth International Conference on*, june 2012, pp. 290 –297.

[8] V. Chifu, I. Salomie, A. Riger, and V. Radoi, "A graph based backward chaining method for web service composition," in *Intelligent Computer Communication and Processing, 2009. ICCP 2009. IEEE 5th International Conference on*, aug. 2009, pp. 237 –244.