End-user's Service Composition in Ubiquitous Computing using Smartspace Approach

M. Mohsin Saleemi
Turku Centre for Computer Science (TUCS)
Åbo Akademi University
Turku, Finland
msaleemi@abo.fi

Johan Lilius
Department of Information Technologies
Åbo Akademi University
Turku, Finland
johan.lilius@abo.fi

***Abstract*—This paper presents our architecture and overall process for creating end-user service compositions using smartspace appraoch. We have used OWL-S ontology language to describe the service capabilities semantically. We implemented the composition algorithm as the planning strategy for automatic service composition. This composition conforms to semantic graph-based techniques where atomic services are composed iteratively based on OWL-S service properties. We also presented a concrete example to show how this algorithm automatically discovers and composes services in a sequence that fulfills end-user's requests.**

***Keywords*-smartspace; ubiquitous computing; composition.**

I. Introduction

Recent advances in information and communication technologies have made available a wide range of devices and services to their users and hence making a device, service and information rich environment for them. It helps simplifying and managing our complex lives, e.g., the ubiquitous smartphone that manages our calender, contacts and task list and helps us keep our lives organized, or the Personal Video Recorder (PVR) whose time-shift functionality allows us to watch TV programming when we want, not at the time prescribed by the broadcaster. However each of these devices is basically an island, with no proper connectivity between the applications. In order to take full advantage, devices need to interact with each other to perform different tasks. The problem thus is the closed and proprietary device architectures which have limitations in terms of scalability and interoperability. By exposing the internal data and functionality of the devices and ensuring interoperability of data, a whole new universe of applications will be possible. For example, your smartphone could notice that your favorite program will start in 5 minutes, based on your profile information or a fan page on facebook and the TV guide available on the broadcaster's web page. Then, it could use GPS to find that you are not at home, and deduces that it needs to start the PVR at home. Another example could be the composition of available services to form a complex composite service which is not otherwise possible.

To enable these kinds of cross-domain scenarios, there are many technical and conceptual problems to be solved. One way to address these issues is through the notion of a *smartspace*. A Smartspace is an abstraction of space that encapsulate both the information in a physical space as well as access to this information allowing devices to join and leave the space. In this way, smartspace becomes a dynamic environment whose membership changes over time when the set of entities interact with it to share information between them. For example, communication between the mobile phone and the PVR in the above scenario does not happen point-to-point but happens through the smartspace whose members are the mobile phone and the PVR. These device functionalities are available to the other elements of the members of the smartspace as services.

We have developed a prototype service composition architecture that supports service composition in smartspace environment. The system is able to achieve the desired user-tailored goals by composing the combination of available services in the smartspace. As part of our solution, we introduce a composition algorithm that find a set of candidate services which could be part of the composition. The complete realization is obtained by grounding of the selected services.

The rest of the paper is structured as follows. Section II describes the smart-m3 which is the underlying architecture of our work. In Section III, we presented the application development approach for smart-m3. This section also proposes our concept of *service* to the smart-m3 architecture and provides an example to illustrate the idea. In Section IV, we specify the service description language and the service composition. We then present our proposed system architecture of service composition in Section V. Section VI illustrates the implementation details of the example scenario of service composition using the language translation service. Section VII presents related research and we conclude the paper and give directions for the future work in Section VIII.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-124-3

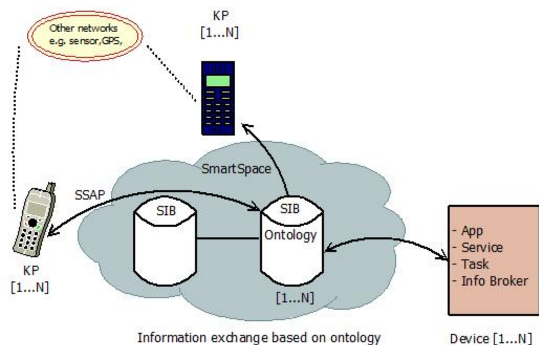

Figure 1. Smart-M3 Architecture

II. Smart-M3 architecture

The Smart-M3 architecture [9][2] provides a particular implementation of smartspace where the central repository of information is the Semantic Information Broker (SIB). The smart-M3 space is composed of one or more SIBs where information may be distributed over several SIBs for the later case. The set of SIBs in a M3 space are completely routable and the devices see the same information, hence it does not matter to which particular SIB in a M3 space a device is connected. The information is accessed and processed by the entities called Knowledge Processors (KPs). KPs interact with the M3 space by inserting, retrieving or querying the information in any of the participating SIBs using access methods defined by the Smart Space Access Protocol (SSAP). Smart-M3 provides information level interoperability to the objects and devices in the physical space by defining common information representation models such as the Resource Description Framework (RDF). In this way, it provides a device, vendor and domain independent solution for interoperability. Since smart-M3 does not constrain to a specific structure of information, it enables the use of ontologies to express the information and relations in an application. The ontology enables the KPs to access and process the information related to their functionality from the M3 space and hence it directs the KPs through the space. Figure 1 shows the overall Smart-M3 architecture.

III. Application development for smart-M3

We chose the ontology-driven application development approach for smart-M3 and developed tools [7] for mapping of ontologies to Object Oriented Programming (OOP). Our approach consists of two parts. The first part is the generator that creates a static API from an OWL ontology. This mapping is done according to a set of static mappings. These mappings generate native Python classes, methods and variable declarations which can then be used by the KP developer to access the data in the SIB as structured and specified in the OWL ontology. The second part is the middleware layer which abstracts the communication with the SIB. Its main functionality to the generated API is triple handling. This consists of inserting, removing and updating triples and committing changes to the smartspace. It also provides functionality for synchronous and asynchronous querying. Our approach enables application developers to use the generated API to develop new KPs and applications without worrying about the SIB interface as the generated API takes care of the connection to the SIB each time an object is created.

In this application development approach, the concept of application is not the traditional control-oriented application running on a single device but the application is constructed from a number of independently operated KPs which may run on different devices and group together to be perceived as a single application. For instance, chat, calendar synchronization and multiplayer games are examples of applications using this approach where a set of KPs each handling a single task run on multiple smart devices and coordinate and interact with each other through the SIB to make a complete application. This coordination between KPs are done in the form of data exchange through the SIB where KPs subscribe to or query for specific data to perform their specified task. Application ontologies are used to describe data in the SIB and directs the KPs to access and manipulate data related to their functionality.

We have taken the following approach to define our system.

The *Knowledge processor (KP)* is a single stateless software entity that reads or writes to the SIB either directly or by subscription. Each KP performs one piece of functionality. The functionality can have different forms and granularities. For example, we can identify device functionality which involves internal device resources such as a KP performing an MPEG encoding function within a PVR to convert analog signal to digital, or it can be a functionality related to computing of user data such as a KP translating a message from one language to another or a KP accessing calendar data in your smartphone. The KPs are accessible by other entities in the system only through the SIB.

The *Application* is the composition of behaviors of the KPs. The application exhibits the total functionality that is of interest to the user. We see applications as possible real-world scenarios that are activated by a particular set of KPs. For example, the scenario of the PVR and the smart phone described in the last section is an application " Record if I am not in home " using our approach. This application is built up by a number of KPs including a set of calendar KPs running in the smartphone and a set of PVR KPs running in the PVR to handle its tasks and functionality. The KPs running in the smartphone includes, for instance, a KP for reading calendar data, a KP for accessing profile information to check for favorite program, a KP for accessing the TV electronic program guide to check the schedule, and a KP for GPS data to find out the current location. The KPs running in the PVR includes, for instance, a KP to turn the PVR

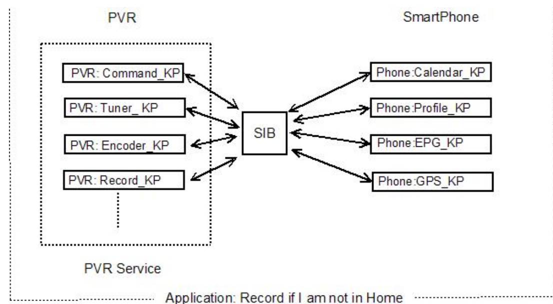

Figure 2. Application Scenario

on or off, a PVR tuner KP that receives the signals, an encoder KP that converts analog signal to digital, a KP to record digital stream to the internal hard drive of the PVR. These sets of KPs in the smartphone and the PVR coordinate by exchanging information through the smart-M3 space to enable this application. From the perspective of an application programmer, the PVR and the smartphone are two the entities in this application that need to be deployed. From deployment point of view, the PVR and the smartphone need to have the capability to tell the smartspace what is available and achievable. In order to make this declaration in a more common and well defined way, the idea of *service* comes in.

We propose to introduce the notion of *service* in Smart-M3 approach as a group of KPs that has a coherent set of functionality. We see a service as an interface to functionality that you can get through the smartspace. For example, the PVR in our example provides recording functionality and has a number of KPs each performing its specified function to implement the overall concept of TV recording. We see this as a PVR service which can be described by a service description language, e.g., OWL-S [1] and then be published in the smartspace to be available for the other KPs that are working within the application. They can call this service and add its functionality in the application without knowing how it actually works. The PVR service can be seen as a form of API where this API can have a number of other PVR functions as well which are not used in our scenario, e.g., you can use the audio-out jacks on the PVR to send the audio portion of programs and movies straight to the home stereo, which may produce better sound quality than the television. This functionality can be implemented by adding additional KPs. The coordination of functionality between the coherent set of KPs is done in the SIB. A single KP can always be seen as a service. The services can be stateless or state-full depending on their functionality. If a service is composed of only one KP that handle one specific task then it will be stateless. In our example application scenario, instead of having individual KPs for handling calendar activities in the application, a set of KPs dealing with calendar data group together to be perceived as a calendar service. These calendar and PVR services are exploited by other KPs in this application, e.g., the KP handling GPS data to check whether it needs to activate the PVR to record a specific TV program. In this way, each service can act as a service provider, exposing its functionality to other KPs and services via the smartspace, and can act as a service requester, incorporating data and functionality from other services. These services are internal to the application, but can be reused by other applications after describing the properties and capabilities of the services using well defined service description languages.

TABLE I
CONCEPTS

Knowledge processor	**Service**	**Application**
Stateless entity	Stateless /stateful	Stateless /stateful
Handles one task	Composition of KPs	Composition of services and/or KPs
Read/write to the SIB	Described by description languages	Represents overall functionality
	Can be discovered and reused	

The SIB offers a persistent data storage back-end and is thought of as a dump and plain database that just gives access to the data. It does not provide any kind of services where some control structure and computations are involved. The coherent set of KPs provide the capabilities to enable services to the SIB in addition to just 'finding' the information. In this way, some computation and control structure is added as the services which have well defined interface and implemented by the set KPs using the object oriented programming approach. By storing the service descriptions in the SIB, it becomes possible to query the Smartspace for its Services. These services can be later discovered, invoked and reused by the other entities in the system. Hence we believe that the notion of service in this context provides advantages to the original smart-M3 approach.

IV. SERVICE COMPOSITION

The basic idea of composition is to use semantically related services in the system in such a way that their combination provides the desired goals which are not otherwise possible. For example, consider a scenario where a number of language translation services are available in a system each translating from one specific language to another. If the system receives a translate service request to a language that is not already available, it should be able to find the services that can combine to fulfill the request. This gives a broader view of service composition that also includes service discovery. The process of service composition is thus, to select a set of services which can possibly fulfill the request (Service discovery process)

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-124-3

and to compose these candidate services to form a single service. All of these tasks are done with the help of service specification languages that describe the services and enable automated or assisted searching of services that participate in the composition process. We propose to use OWL-S language for this purpose. There are several reasons for choosing OWL-S for service description. Firstly, OWL-S enables declarative advertisement of service properties and capabilities that can be used for automatic service discovery. Secondly, as we are using OWL ontologies for the domain concepts and application development for the Smart-M3, OWL-S describes the services in terms of capabilities based on OWL ontologies. Thirdly, OWL-S provides specification of prerequisites of individual services and a language for describing service compositions and data flow interactions. OWL-S can be used to construct complex composite services using OWL-S control constructs such as sequence, split, split-join, if-then-else, iterate, choice and loops. Currently we use only sequence construct in our application which requires a list of components to be done in an order where the output of the component A is compatible with the input of component B and so on.

The OWL-S provides three levels for service description: Service Profile, Process Model and Service Grounding. The Service Profile provides a general description for advertising, discovering and composition of the services. It includes both functional properties of services, IOPE (inputs, outputs, preconditions and effects) and nonfunctional service properties (name, text description, category and additional service parameters). The Process Model gives information about how a service performs its operation and describes the steps that should be done for the execution of the service. These steps include transformation of the set of inputs into the set of outputs and state transitions from one state to another when the service is carried out. The Service Grounding gives details about how to access a service using the specific message formats and platform provided protocols, for example, the Simple Object Access Protocol (SOAP) and HTTP used in accessing web services and the SSAP protocol in our case of the smartspace based applications. As the Service Profile describes the functional description that will be used in the service discovery and composition, we will only focus on Service Profile in this paper.

V. System Architecture

We have proposed a service composition framework for the smart-M3 approach. The architecture of the framework is presented in Figure 3. The goal is to enable both the end-users and the application designers to use the system for service discovery and composition in smart-M3 based application development. The functionalities supported by this framework are publishing the service descriptions in the SIB, searching the services relevant to a query or application based on these descriptions, and composing the compatible services to form a single service to fulfill a given request. The framework consists of the following six layers.

1. User/Application Layer: The goal of this layer is to handle interaction with the end-users. It consists of user level application and the graphical user interface (GUI). The system is expected to receive the request from the end-users in the form-based query mechanism. It can then be converted to the OWL-S representation by the Interpretation layer of the system. The end-user interacts with the system using the application GUI that consists of a user query form. The end-user submits its request by filling the query form and specifies the desired goal in terms of its functional parameters. These parameters would be in the form of inputs, outputs, conditions and goals. This request enables the end-user to specify what he wants the service to do for him while abstracting the implementation of the service. For example, if a user wants to use the system for a language translation service, he submits the request in terms of the source language, the destination language, the goal and/or some precondition by using the end-user query form. The user will totally be unaware if the result of the request comes from a single service or composition of two or several services.

2. Interpretation layer: This layer is responsible for mapping the end-user's intent in some well defined notion, the OWL-S service ontology in our case. After the end-user specifies the desired goal using the application GUI in user/application layer, the user's request is converted into its equivalent form of the OWL-S service ontology by the interpreter agent in this layer. In this way, the end-users do not require to have extensive knowledge about the ontologies as the system handles their requests in natural language. The service descriptions in the request is then matched with the available services in the system to find out if the request can be fulfilled. This layer also contains a composition KP which is responsible for making the compositions of the available services if the request is not fulfilled by a single service. This composition KP relies on the Service Discovery component in the Semantic Service Layer. The Service Discovery component selects a set of services from the available services and passes this information to Service Composition which generates the composition by specifying the order in which each service is to execute to form a single composite service to fulfill the request. As the system is capable of interpreting OWL-S, the applications can also be given as OWL-S.

3. Semantic Service layer: The semantic Service layer handles the description, discovery and publishing of services. End-users can use the system to find out if it is able to do a particular thing such as translating from one language to another. There is another broader way in which services are written in OWL-S and need to be interpreted and executed by the system. For example, the PVR service and

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-124-3

the calendar service are provided by their service providers in OWL-S and the end-user application 'Record if I am not in home' interpret and execute these predefined OWL-S services in the system. As described in Section 3, a coherent set of KPs makes a service, e.g., the PVR service includes several KPs each performing a single task. Moreover, each KP handling a single task can also be seen as a service, e.g., each KP translating from one language to another is regarded as a separate service. In order to make these services available to the other entities in the system and to facilitate service composition, each of these services must be described using OWL-S. The semantic service layer is responsible for these service descriptions. After describing each service, the services need to be published. The service provider describes service functionality and other information which must be converted in OWL-S in order to store and publish in the SIB. The Service Discovery is done after services have been described and published and return a list of services as a result to the composition KP.

4. KP Layer: The KP layer is the low level layer which is responsible for bindings of the KPs. Each KP handles one task or functionality and interact with the repository to read and write required data. An end-user application contains a series of actions that need to be performed to fulfill its desired goal. These actions are actually implemented by the KPs in the system.

5. Middleware Layer: The middleware layer contains all the necessary components that are required to access the devices and services in the smartspace. It includes DIEM Mediator that provides a smartspace independent interface for accessing certain functionality of the SIB. This interface encapsulates the communication of KPs with the SIB and provides modularization. It provides the functionality of triple handling and synchronous and asynchronous querying. The middleware layer may optionally include other 3rd party middleware solutions such as UPnP middleware.

6. System Layer: The system layer consists of the SIB which acts as a persistent data storage for the information in the form of RDF triples. This layer also contains the operating system and other local device resources and provides a native interface to the upper layers.

In this system we are proposing the OWL-S language as a kind of interoperability format for KP-based smartspace applications. The notion of service in this kind of applications gives a well defined interface which provides benefits over traditional Smart-M3 application development approach which is based on only KPs. These benefits include, for instance, description, discovery and composition in a more structured way.

By stroing the service descriptions in the SIB, it becomes possible to query the smartspace for its services. From a programming point of view, the notion of services is a structuring mechanism that groups together several different KPss to abstract individual calls from the API of a device

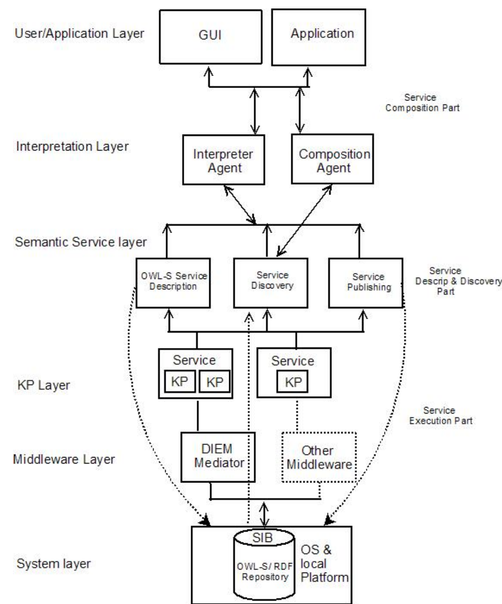

Figure 3. Structure of the System

that is present in the Smartspace. We make a dedicated space in the SIB, that can be used for registering the services. In this case the SIB will be the central part and all the requests and responses of the services will go through the SIB. For example, when a service requester submits a query to the SIB, the SIB will interpret the query and determine the match using the semantic description of services. After selecting the matching service it will construct a query to the service provider agent. The service provider agent interprets this query from the SIB, performs the service execution and generates a response to the SIB. The SIB interprets this response, transforms it to the response understood by the requester and sends it to the service requester.

VI. Example Scenario: Language translation

To illustrate the approach, we consider the following service composition scenario where many KPs in the system coordinate each other to fulfill the end-user's request that is not possible with a single KP. The scenario is the language translation function which translate from one source language to the target language. Suppose we have many KPs in the system where each is capable of translating one particular language to another. We call each of the translate functions as a Service. An end-user uses the system and sends the query to translate a particular message in a source language, for instance, Finnish to a target language, for instance, Urdu. Since each KP has to perform its own dedicated single task, there is no KP that directly satisfies this request. In other words, there is no single service in the system that can perform this task. In this case, the system is able to find appropriate composition of different

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-124-3

services to satisfy the user's request. The system performs this composition automatically hence the end-user does not need to worry about invoking each individual service that takes part in the composite service.

A. Composition Algorithm

We have implemented a composition algorithm for service composition. The algorithm is implemented in python as we have only a python version of the SIB interface available at the time of writing this paper. The algorithm follows the Breath First search pattern to find all the candidate services that can be included in the composition. The Breath First search gives better results as it moves horizontally across each search path and finds the solution with the fewest possible services and does not face looping problem. As the SIB can read and write only RDF data, our algorithm use the same format to interact with the SIB to insert and query the RDF triples. The algorithm works in two parts. In the first part, the services which can be included in the composition are separated. In the second part, the composition is constructed based on the selected services using the approach that begins with inputs and preconditions and works forward in the direction of outputs and effects. This shows that each and every service should be described semantically in terms of its inputs, outputs, preconditions and possible effects in order to make this algorithm work. Our algorithm always finds a composition solution if it is available unless there are infinite services. If the composition is not possible from the available services in the system, it returns the message that the user's request cannot be fulfilled. The proposed algorithm can be evaluated based on its completeness and optimality. The time and space complexity depends on the particular service compositions. As our focus in this project are the smart devices, we have implemented the algorithm in a way that reduces the space complexity by ceasing the algorithm at some depth to avoid memory overflow. However, there will be compromises in terms of completeness because the algorithm may not find the solution within that search path.

When an end-user requests a specific service by making a query, the algorithm starts with checking the inputs and preconditions of the requested service and matching these parameters with the available services. All the services which have similar inputs and precondition parameters as the requested service may be possible candidates for service composition and are selected. The algorithm then works by constructing a graph of the services in a forward chain pattern by checking the outputs and effects of the previously selected services. This process continues until it reaches to the service having desired output. This service is placed at the leaf node of the graph and the composition is returned by traversing the graph.

B. Example

We use the example of language translation to illustrate the applicability of the proposed algorithm. We assume that the system has a range of different services provided by indiviasual KPs where each KP performs only one task. An end-user wants to use the system to send the meeting invitations to all members of the project in their mother tounge. The original invitation is in the Finnish Language. The user sends the request to the system by using some GUI to translate the particular Finnish text into the desired language, for example Urdu. Suppose that there are many different KPs, each providing a translation service from one particular language to another. The Service handler takes this request and prepares the semantic description of this request in terms of inputs, outputs, preconditions and effects. In general, Inputs and Outputs are subclasses of a general class Parameter in the OWL-S service ontology. Every parameter has a type that can be specified using a URI to uniquely identify it. The type can either be a Class or a Datatype such as a number, a string etc. The Preconditions and effects represent more specific functional properties to easily discover the services. For example, assume the service request is to translate from a source language to a particular target language with the input and the requested output of type string. There might be services in the system that have the same input/output type but their goals and preconditions are different, such as a dictionary service which has precondition of the same input and output language.

This information is derived from the service profiles of each of the available services. The profile gives the details about each service. The following OWL-S statements shows the general profile of the language translation service.

```
<profile:Profile rdf:ID="Lang">
<profile:serviceName
rdf:datatype="http://www.w3.org/...#string">
Lang Trans
</profile:serviceName>
<profile:hasPrecondition
rdf:resource="#Lang_precond"/>
<profile:hasInput>
<process:Input rdf:ID="message">
<process:parameterType rdf:datatype=
"http://www.w3.org/...#anyURI">
http://...#Message
</process:parameterType>
</process:Input>
</profile:hasInput>
....
</profile:Profile>
```

These parameters are then matched with the available services in the system to find out if there is any single service that matches all these parameters. If there is any, a message is sent to its respective KP to execute the service using the text as input and the results are returned to the end-user. If there is no such match found, then the composition algorithm starts with finding all the services which has

the same input, preconditions and effects as the requested service i.e the services accepting Finnish language as input, having different input and output languages as preconditions and translation as the goal/effect. These services could be the possible candidates for the composition and hence separated by constructing a graph with each node representing a service in the same horizontal level. Suppose that the system finds the following services in this step.

```
S1: <Input:Fin><Output:Spa><Effect:translation>
S2: <Input:Fin><Output:Eng><Effect:translation>
S3: <Input:Fin><Output:Ger><Effect:translation>
```

The algorithm constructs the graph by placing 'Fin' at the root and Swe, Eng and Ger as its immediate children. It then checks each immediate children by checking their outputs and effects and putting each matching services under their respective nodes. Suppose the following services are retrieved in this step.

```
S4: <Input:Spa><Output:Eng><Effect:translation>
S5: <Input:Ger><Output:Dek><Effect:translation>
S6: <Input:Eng><Output:Urd><Effect:translation>
S7: <Input:Eng><Output:Fre><Effect:translation>
S8: <Input:Eng><Output:Swe><Effect:translation>
S9: <Input:Ger><Output:spa><Effect:translation>
```

The algorithm then checks if the composition is possible from the selected services as the original requested output (Output: Urd) is found in this step. It traverses the graph to reach the root (Input: Fin) and finds the path. The resulting composition S2->S6 is returned to the service handler. It is important to note that based on the selected services, there is another composition possible for the same request <Input:Fin ><Output:Urd >which is S1->S4->S6. The algorithm always returns the composition which involves fewer services. If the service composition is not found in this step, the algorithm continues working until it reaches the desired output or there is no composition possible from the available services. For the later case, the algorithm returns a message of 'composition not possible' to the service handler. The algorithm can run indefinitely if there is very large number of services and it does not find any suitable composition to fulfill the request. This means that we need to apply some end point to limit the number of times it executes to reduce the memory and bandwidth requirements of smart devices. Figure 4 represents the results of the service composition using our algorithm.

The composite service produced by the composition of atomic services can then be described semantically so that it can be discovered or take part in other compositions later. This way, we are able to provide services that are not actually included in the system.

C. Service Grounding and Execution

After the service composition is created, the next step is to execute these services in order to get the desired goals. The service grounding prescribes the details of how to access

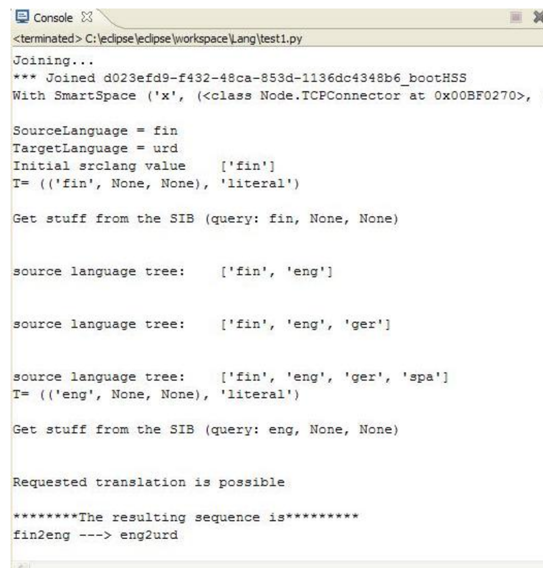

Figure 4. Composition results

the services in the composition such as message formats, protocols and invocation methods etc. used to interact with the composed services. It specifies a link between semantic and non-semantic description elements of the services. The grounding can include some or all of the following elements

```
Non-semantic elements:

ServiceName -> Name of the service
ServiceType -> Type of the service
KPInfo -> Information of KP providing service

Semantic elements:

ServiceInput -> Input of the service
ServiceOutput -> Output of the service
InputType -> Type of the Input parameter
OutputType -> Type of the output parameter
Preconditions -> Conditions met before execution
ServiceEffects -> goal of the service
MessageFormat -> Format acceptable by KPs
Protocols -> protocols to interact with the SIB
ServiceOntology -> specifies relations
```

As in our system implementation, every co-ordination between the KPs goes through the SIB, the execution of the services in the composition is done by invoking each individual service and passing data between the services in the order specified by the composition using the grounding elements described above. This invocation is accomplished by sending the messages to the service provider agents in their acceptable formats. This approach can undergo scalability problems in case of an excessive message and data passing between a large number of services in the composition such as in sensor applications.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-124-3

VII. RELATED WORK

There are different approaches and architectures that address the issue of service composition. These approaches can be classified using several service composition features such as automatic composition [8], semi-automated composition [10], end-user interaction [11] and service specification lanaguage [6] etc. In [3], the authors give a comparison of different service composition approaches. A middleware solution for end-user application composition is provided in [5]. Other approaches of flexible service composition in mobile environments are described in [4][12]. While existing research efforts deal with theses issues separately, there has been very limited work in ubiquitous service compositions in smart environment. In [13], the authors proposed a system consisting of a middleware and user-level tools that enable the end-users to combine, configure and control the services using their smart home devices. Each home device has interfaces and the end-user selects some devices and the system generates a set of compositions of selected devices in a way that these devices can collaborate to generate applications by using the domain knowledge and user inputs. This composition may result in an arrangement that has no meanings. They have used the Depth First Search (DFS) algorithm for generating the composition. The drawback with this algorithm is that it does not know which composition is better and which composition does not make sense. In our approach, each service is described semantically and this service description is stored in the SIB. It gives the advantage of query and accessibility of the services from the smartspace.

VIII. CONCLUSION

In this paper, we expressed our ideas for providing services to the smart-m3 approach. These services are provided by means of sets of Knowledge Processors. We presented the system architecture and example services to illustrate our approach. A service composition algorithm is also presented with a concrete example. For future work, we are aiming to implement an efficient algorithm for discovering the services in the big search space of smart-m3. Furthermore, we need to use the ontology hierarchy to restrict the set of services considered for matching.

ACKNOWLEDGMENT

The research work presented in this paper is based on the ICT-SHOCK DIEM (Devices and Interoperability Ecosystem) project and the authors would like to acknowledge all the partners of this project.

REFERENCES

[1] Owl-s services: http://www.daml.org/services/owl-s/. Accessed: October 2010.

[2] Smart-m3 software at sourceforge.net, release 0.9.4beta, may 2010. [online]: http://sourceforge.net/projects/smart-m3/. Accessed: October 2010.

[3] J. Bronsted, K. M. Hansen, and M. Ingstrup. A survey of service composition mechanisms in ubiquitous computing. In *Ubicomp 2007, pages = 87-92*.

[4] D. Chakraborty, A. Joshi, T. Finin, and Y. Yesha. Service composition for mobile environments. *Mob. Netw. Appl.*, 10:435–451, August 2005.

[5] O. Davidyuk, N. Georgantas, V. Issarny, and J. Riekki. MEDUSA: Middleware for End-User Composition of Ubiquitous Applications. In *Handbook of Research on Ambient Intelligence and Smart Environments: Trends and Perspectives*. IGI Global, 2010.

[6] J. Dong, Y. Sun, S. Yang, and K. Zhang. Dynamic web service composition based on owl-s. *Science in China Series F: Information Sciences*, 49:843–863, 2006. 10.1007/s11432-006-2026-2.

[7] A. Kaustell, M. M. Saleemi, T. Rosqvist, J. Jokiniemi, J. Lilius, and I. Porres. Framework for smart space application development. In *Proceedings of the International Workshop on Semantic Interoperability, IWSI*, 2011.

[8] S. Majithia, D. W.Walker, and W.A.Gray. Automated web service composition using semantic web technologies. In *Proceedings of the International Conference on Autonomic Computing (ICAC04)*.

[9] I. Oliver and J. Honkola. Personal semantic web through a space based computing environment. In *Proceedings of the 2nd IEEE International Conference on Semantic Computing*, 2008.

[10] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic composition of web services using semantic descriptions. In *In Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003*, pages 17–24, 2002.

[11] Z. Song, Y. Labrou, and R. Masuoka. Dynamic service discovery and management in task computing. In *Mobile and Ubiquitous Systems: Networking and Services, MOBIQUITOUS 2004.*, 2004.

[12] M. Valle, F. Ramparany, and L. Vercouter. Flexible composition of smart device services. In *The 2005 International Conference on Pervasive Systems and Computing(PSC-05), Las Vegas*, pages 27–30, 2005.

[13] P. Wisner and D. N. Kalofons. A framework for end-user programming of smart homes using mobile devices. In *Proceedings of the Consumer Communications and Networking Conference, CCNC*, 2007.