Usable developer-oriented Functionality Composition Language (UFCL): a Proposal for Semantic Description and Dynamic Composition of Services and Service Factories

Remi Emonet, Dominique Vaufreydaz

LIG - PRIMA - INRIA Rhône-Alpes
655 avenue de l'Europe, Montbonnot, 38334 Saint Ismier cedex, France
{Remi.Emonet,Dominique.Vaufreydaz}@inrialpes.fr

Keywords: SOA, semantic description, service factories, dynamic and abstract composition

Abstract

This paper presents a Usable developer-oriented Functionality Composition Language (UFCL) designed for ubiquitous systems developers. Easy to write, this language is used to semantically describe functionalities implemented by services in a service oriented architecture where each service exposes its own description. Service factories can also be described using UFCL: a factory defines an abstract composition pattern and is able to instantiate product services on demand. This paper also describes UFCL compilation that makes it possible to reason about functionalities exposed by services and factories.

1 Introduction

In the last sixty years, we went from one computer on earth to a one to one relationship between computing devices and people. The number of personal computing devices continued to increase: cellular phones, mp3 players and personal assistants; but also devices spread into the environment such as video camera, motion detectors or microphones. Given the apparition of wireless networks, all those devices can communicate together. Introduced by Marc Weiser [13], this vision where computation enabled devices are present everywhere and can communicate together is called "ubiquitous computing" or "pervasive computing". Building upon this network of devices, ambient intelligence tries to make these devices address the user in an appropriate way by making them aware of its activity: current task, availability, current focus of attention, etc. These environments that sense user activity and act according to it are named "intelligent environments" or "smart environments".

Major constraints concerning these intelligent environments are that such systems are always running. New devices can come in and present devices can leave or break; this leads to a highly dynamic environment. Additions and removals can be really frequent: imagine an exposition hall where people enter and leave all the time bringing in and out their cellular phone, their laptop, etc. To properly interact with human users, applications must be aware of their environment and its modifications. An application cannot make any supposition about sensing or acting device availability or about computational service availability.

2 State of the Art

A classical architectural solution used to solve constraints of pervasive computing and intelligent environments is to adopt a *Service Oriented Architecture* (SOA). Basic concepts of SOA are to build applications by using dynamically discovered services. In this context, a service is a piece of software with a well-defined way to access the provided functionality. In SOA, implementation is separated from the way to access functionality. A service can be used without bothering to know how it is implemented. Services can thus be designed with low coupling and in an abstract manner allowing reuse, interoperability and substitution of a service by another.

Web Services technology is probably the main representative of SOA. It implements SOA by leveraging Web technologies: it uses XML message over HTTP, Simple Object Access Protocol (SOAP) and Remote Procedure Call (RPC). Many extensions are built upon it for example to handle security or binary attachments. Web services expose their access points using a dedicated XML language: Web Services Description Language (WSDL). Accessing a web service is done by retrieving its WSDL description that lists access points on which HTTP request can be sent. The consumer of the service does not know anything about service implementation and is properly decoupled from the underlying implementation. It still have to know the communication protocol in a wide sense: names of the methods to call, signatures, valid call sequences, etc.

Semantic web services try to improve decoupling: not only a service consumer does not have to know anything about a service implementation but also it does not have to know exact method names and signatures. To achieve this goal, services exposes a set of capabilities, defined in an ontology expressed in Web Ontology Language for Services (OWL-S). OWL-S brings semantic description and reasoning to service oriented architecture but is not well suited for abstract composition patterns description [11, 4]. Describing a semantic web service consists in listing its exposed semantic capabilities and how they are grounded in the implementation. Grounding is represented by a reference to

a WSDL access point joined with a transformation to apply to a semantic query to convert it into a query understandable by the access point. Such transformations are expressed using a common language: Extensible Stylesheet Language Transformations (XSLT). Semantic descriptions of capabilities exposed by services increase interoperability and allow spontaneous interactions. Ontology concepts can be marked as equivalent: for example, it can be expressed that LanguageSwitcher and Translator are two equivalent concepts. A service consumer looking for a Translator can reason about this equivalence and use a LanguageSwitcher instead. This permits spontaneous interaction between services not originally designed to work together. Automatically building concrete adapters from one concept to another is a research field by itself [5, 14] and we are not trying to tackle this problem.

Composing existing services to create new ones also improves flexibility of systems and spontaneous interactions. One can choose a set of existing services and wire them together to obtain an implementation of a new service. For example, given two translators, french to spanish and spanish to english, we can expose a new translator from french to english by composing them. This is simple composition but we can also have abstract composition patterns. In the example, we could have a translator composer that would express its ability to compose any two translators (with some constraints) to give a third one (with some properties). Typical existing works in this research field are languages such as Business Process Execution Language (BPEL) and its adaptation to web services: BPEL for Web Services (BPEL4WS). One can use such language to describe how to assemble existing services, detailing how to orchestrate existing services to implement a new functionality. BPEL4WS is designed to express a sequence of calls to web service methods but it works at a non-semantic level (WSDL descriptions) and is not designed to interconnect services making them exchange stream of data [10].

3 Proposal

We want to go further than BPEL4WS and have *both semantic description* of service functionalities and *abstract service composition patterns*. We want to jointly reason about semantic description of service functionality and possible composition patterns like in BPEL.

Very recent work on extending BPEL4WS with semantic capabilities can be found in [12]: keeping compatibility with existing Web Service technologies, they make service description abstract and semantic. They replace WSDL in BPEL descriptions making it possible to describe abstract composition patterns. It is, however, not clear whether their implementation is covering all their exposed concepts and whether they are doing extensive inference based on composability.

In the component-based design community, most projects have an implementation and it is usually open source. OSGi [1] now provides a service-oriented environment based on components; it integrates works presented in [6, 7]. OSGi Service Component Runtime (SCR) uses classical Java interfaces and some properties to represent service functionalities. Using SCR, one can easily describe what functionalities a component implements and requires. The component is automatically notified of dynamic availability of its requirements. OSGi runs on a single Java Virtual Machine (jvm) but work presented in [3] extends OSGi and SCR to run across multiple jvm. However OSGi SCR is simple, powerful, fully implemented and robust, it still missed semantic description of functionalities.

In our intelligent environment, we have many sensors emitting textual or binary data that can be important (sound, video, laser measures, etc.). Web services are not designed for streaming communications and it is one reason why we use OMiSCID [9], a middleware implementing SOA while allowing efficient transfer of massive binary data. Using OMiSCID, one can declare services and search for services using basic AND/OR/NOT combined requests, interconnect them, etc. Another constraint is that we target developers hardly knowing XML and web technologies: asking these users to learn OWL, XSLT or BPEL would be a complete failure. We propose to bring these technologies to non-specialist users by providing them with a simple language to express these concepts: the Usable developer-oriented Functionality Composition Language (UFCL). UFCL is also specially designed to allow reasoning about possible compositions.

Our *contribution* through this article is threefold:

- we propose the Usable developer-oriented Functionality Composition Language (UFCL) : a **simple language** that aims at semantically **describing services** and more originally **service factories**.
- we implement a **compiler** from UFCL to an existing rule-based system that makes it possible to do **inference on available functionalities**.
- we illustrate how to use such tools to have a better and more interoperable architecture in **intelligent environments** using a **concrete example**.

In section 4, we expose UFCL: its syntax, semantic and expressive power. Section 5 is dedicated to an overview of how UFCL descriptions are compiled to some rules and facts in Jena, a RDF-based rule system. Section 6 gives an example of how an intelligent environment application can be rearchitectured using the proposed approach to minimize coupling between different software components and allow integration of services not explicitely designed to operate together.

4 UFCL Principle and Examples

The goal of the Usable developer-oriented Functionality Composition Language (UFCL) is to provide the developer with an easy way to express three kinds of knowledge. First, one expresses what semantic functionalities are implemented by a service. This enables further service discovery based on semantic functionalities. Second, one expresses correspondences between functionalities. This

is actually useful to integrate services from other developers/vendors by adding correspondences between functionalities they provide, and functionalities other services are looking for. Third, one describes which products a service factory can instantiate. A service factory is a service with the particularity that it can instantiate new services on demand/need. The concept of service factories appears in existing works such as [8] where CORBA is extended to provide easy object migration. Factories are deployed on multiple host but they only create generic blank objects into which the migrating objects' state is pushed.
Details and examples concerning these three kinds of description are given in the following subsections.

4.1 Describing Service Functionalities

The most basic need for semantic description is to express what functionalities or capabilities are provided by a running service. Once provided functionalities are exposed, it becomes possible to look for services based on required functionality instead of service identity or method signature.
OMiSCID middleware only exposes services with a unique identifier, a list of variables and a list of connectors. Specific variables are present in any service: a name, an owner and a "class". However "class" variable could be used to express the functionality implemented by a service, it is intended to hold only a class name. As we want to possibly implement multiple classes, we cannot restrain to using this variable. Thus, we use a custom *knowledge* variable containing the UFCL description of the service. Using OMiSCID, every service can dynamically contribute or retract UFCL knowledge and any client can gather this knowledge and infer possible compositions (see section 5).
UFCL aims at being simple to write: we made the supposition that basic concepts of object-oriented programming are now known from most developers and we based our language on those concepts. UFCL allows a service to implement a class defined by its class name and by a set of valued properties, in the same way that an object has a given type with some values for its fields.
Here is a simple example of UFCL expression that defines such a functionality implementation. Line numbers have been added to allow us to reference it from the comments following this UFCL snippet:

```
1  namespace is http://ie08#
2  this isa Timer
3    with freq = 2
4    with grounding = "C(tickTwice)"
```

This UFCL expression simply declares that the current service is exposing a *Timer* functionality with a frequency of 2. As a first remark on UFCL syntax, we can precise that whitespaces and indentation are unimportant as they are in most of the programming languages.
In this declaration, line 1 gives a default scope for semantic names such as *Timer* at line 2. Given the namespace declaration, *Timer* will be interpreted as `http://ie08#Timer`. Our language allows writer to use multiple namespaces and give them aliases using a syntax inspired by classical XML namespaces (e.g. *myNS:Timer*). At line 2, *this* is a special identifier that implicitly represents the service exposing the knowledge. Identifier *this* could have been equivalently replaced by the exposing service identifier. Using *this* is a convenience for service designer as it allows instance-independent knowledge to be written. It is helpful not to use *this* in the case where a service describes another one. The need for describing another service may arise when a service has been deployed with incomplete or no description.
Still at line 2, UFCL keyword *isa* introduces the functionality or capability implemented by *this* service. In this context, we allow one service to implement more than one functionality. We named this implementation a functionality facet: in the current example, we can say that *this* has a *Timer* functionality facet. One can note that *isa* could be replaced by *implements* which would be semantically better; however, in object-oriented design and languages, the implementation is associated to interfaces or purely abstract classes which are concepts that not all developers fully master. At line 2, *Timer* (standing for `http://ie08#Timer`) is simply a functionality name that may but does not need to be used or defined elsewhere. In fact functionality names are Unique Resource Identifiers (URI) like the one used in Resource Description Framework (RDF) and Web Ontology Language (OWL). This is intended to keep the door open to reuse possibly existing ontologies when describing implemented functionalities.
Lines 3 and 4 affect values to two properties of the functionality facet, *with* being only a separating keyword. In this example, one of the two properties (*freq*) is a simple property of the *Timer* functionality facet. The other does not particularly concern the timer facet. The special *grounding* property attaches the defined functionality facet to an existing software service running in the environment. In this case, it tells us that the *Timer* facet is implemented using the single connector called *tickTwice* to be found on the service exposing the knowledge. Here, the grounding implicitly refers to a connector on the service implementing the functionality facet, *this* in the given example. Rare are the cases where one functionality facet should reference variables and connectors from services other than the facet owner but it is still allowed by UFCL grammar. To underline the semantic of *isa*, we can imagine that our service, that has been declared to be a Timer with a frequency of 2 is also a Timer with frequency one, tick events being sent on a connector named *tick*. In this case, we would have another Timer facet declaration for the same service:

```
5  this isa Timer with freq = 1
6    with grounding = "C(tick)"
```

This would not be permitted by a simple class belonging or classical interface implementation principles.

4.2 Describing Functionality Correspondences

Having described functionalities exposed by our services makes it possible to do service selection based on these functionalities but it is still required to have an a priori

agreement on what functionalities will be used. Reconsider our *Timer* functionality introduced before: it only sends events at a fixed rate. In a different context, for example in a music oriented application, the designer could have named this functionality *Metronome*. We would like an application designed to work with *Timer* to spontaneously be able to use a *Metronome* service that would be present in the environment. To allow this integration, two kind of knowledge are required: the knowledge that *Timer* and *Metronome* are equivalent concepts (modulo the name and unit of frequency) and the knowledge of how we can convert metronome's messages and communication protocol to timer's. These are two concerns that can be separated and we concentrate on the first one.

Integrating a metronome in an application that expects timers requires a knowledge about this concepts correspondence. Either written by a human or generated using an automatic method, this knowledge is mandatory. UFCL proposes a way to describe this equivalence between functionality facets. Here is an example:

```
 7  a Metronome
 8      having bpm = ?f
 9    isa Timer
10      with freq = ?f / 60
```

This UFCL fact expresses that a *Metronome* functionality facet can be used as a *Timer* functionality facet. Lines 7 and 9 put the two concepts in relation while line 10 tells that the *freq* property of the produced timer has to be set to "?f / 60". *?f* is a wildcard defined in line 8 as having the value of *bpm* property in the Metronome facet (bpm stands for beats per minute) and has to be divided by 60 to be converted into a frequency with the unit expected in Timer facets. Globally these 4 lines state that any service having a *Metronome* facet (with a *bpm* property but no particular constraints) will also have a *Timer* facet with its *freq* property set to 1/60th of *Metronome*'s facet property *bpm*. Correspondences between functionalities can be seen as a kind of ontology alignment, putting into relations concepts introduced by different designers. The *grounding* property that is defined for all functionality facets is automatically propagated in case of functionality correspondences. In this example, the *Timer* facet will inherit from *Metronome* facet's grounding.

The second kind of knowledge required in functionality correspondences concerns message format and communication protocol adaptation. It is not described by UFCL functionality correspondence. However it is an important problem that drives many research efforts, we do not handle this part of format and protocol adaptation, we just provide this clear architectural separation between description of semantic functionality correspondences and adaptation of communication formats. In our case, formats and protocols are stored in service description: OMiSCID middleware associates to each service connector or variable a description and a format description. Using this middleware, these descriptions are the base information for protocol and format adaptation. In a fully operational system, we could allow services to expose knowledge about protocol and format adaptation, for example using text manipulation scripts and/or XSLT as it is done in OWL-S. In addition to these simple descriptions, we should also allow dedicated services to convert messages on demand (format and protocol adapter services).

4.3 Describing Factories and Abstract Functionality Compositions

Both constructs presented in the previous sections are classical in existing semantic service description using ontologies. Through a simple syntax, UFCL aims at bringing these concepts to the average non semantic web specialist developer. In this section, the mechanism presented is more original and consists in describing service factories. The role of a service factory is to instantiate other services on demand.

There are mainly two kinds of factory for two kinds of instantiations:

- parameterized service instantiation: given desired parameters, we can produce a service instance with those parameters. Our Timer facet is a good example of this kind of instantiation; we could easily write a factory that would instantiate any new Timer service given the desired frequency.

- composite service instantiation: given some references to existing services we can produce a new service by composing their functionalities. We can illustrate this by an example with translators. We could write a factory that would use any Translator from language A to language B together with one from language B to language C and produce a Translator from A to C.

One can imagine having a factory that both uses one or more services and accepts some free parameters.

We claim that factories are a necessary extension to existing service oriented architecture that uses service repositories. The timer/metronome example is underlining this point: it would be impossible to advertise the presence of all possible timers as there is an infinity of them but factories allow to instantiate any required timer. Factories have to advertise what family of services they can instantiate and under which conditions.

UFCL handles declaration of service factories we just presented. A first example is the one concerning *Timer* factory service that would expose this UFCL description:

```
11  composing
12    grounding "C(start)"
13    format "<run f='{?pFreq}'/>"
14  gives a Timer
15    with freq = ?pFreq
```

Lines 14 and 15 express the fact that this factory produces services implementing *Timer* functionality and that *freq* property takes the value of *?pFreq* wildcard. No constraints are expressed on this *?pFreq* wildcard in the rest of the expression. It is completely free and any *Timer* can be produced. Any service consumer understanding this UFCL expression and requiring a Timer with a specific frequency

would be able to ask the factory for the needed *Timer*. First section of the expression, after *composing*, sets some properties for the factory and may define required facets for composition (none here). While *grounding* factory property works like its homonym in functionality facets, *format* factory property tells the factory client how to format instantiation requests. In this example, to ask for instantiation of a Timer having a frequency of 7, one should send on connector *start*, a message like "<run f='7'/>".
An optional section that is not present in this example allows the factory to express constraints on used wildcards and required facets. This section is illustrated in the following example on *Translators* composition factory:

```
16  composing
17    grounding "C(start)"
18    format "<c a='{?t1#}' b='{?t2#}'/>"
19    a Translator ?t1
20    a Translator ?t2
21  having
22    ?t1.to = ?t2.from
23  gives a Translator
24    with from = ?t1.from
25    with to   = ?t2.to
```

This example contains many references to two wildcard *?t1* and *?t2* defined at lines 19 and 20. These two wildcards are each representing one *Translator* functionality facet. At line 22, in the *having* section, is defined the only constraint between these two functionality facets: to be composable, destination language in *?t1* must be the same as source language in *?t2*. As in the previous example, last section at lines 23 to 25 expresses which functionality this factory produces. These lines tell us that this factory can produce a new Translator with the same source language as *?t1* and the same destination language as *?t2*. This example is particular and uses three times the same functionality (*Translator*) but any functionality can be used. At line 18, format uses "?t1#" and "?t2#" to express references to the grounding of functionality facets. In our case, services are represented by some unique numeric identifier and a formatted message accepted by this factory could look like "<c a='C(1234:tr)' b='C(5678:tr)'/>" where "C(1234:tr)" references *tr* connector (translate) on service with identifier "1234".
Factories receive messages containing only grounding information and no facet references. Grounding information can be easily interpreted by a service without any knowledge of UFCL and existing facets. This is made to decouple the factory implementation layer from UFCL. One can build a factory service or use it without any exposed UFCL description. These examples do not feature combination of facet wildcards (like *?t1*) and parameter wildcards (like *?pFreq*) but UFCL allows such combinations.
Systematically advertising and looking for services with semantic functionalities together with the presence of service factories bring a lot of flexibility and awareness to dynamic change in software environment. One can easily integrate into an already running application an alien service and have it interoperate with existing services. However, to make this spontaneous interoperation come to life, we must be able to manipulate and reason about all these semantic descriptions. In the next section, we present a compiler from UFCL to some rules that makes it possible to take advantage of the presented constructs.

5 UFCL Compilation and Reasoning

This section is devoted to how we effectively transform UFCL knowledge to enable reasoning. It gives an overall presentation of the compilation from UFCL to a rule-based system making it possible to do powerful reasoning. Everything presented here is *implemented and operational*.

5.1 Functionality Facets Representation

Given the nature of reasoning that needs to be done on the previously presented knowledge, we preferred using a rule-based system over a fully handmade dedicated algorithm. Reusing existing rule-based system cuts down development time and offers some guaranties about robustness, performances and maintainability. We chose to use Jena [2], an opensource and well designed rule-based system dedicated to semantic web. Jena's grounding into semantic web makes it uses a set of RDF triples as its knowledge base. Jena allows both forward and backward rules to be expressed. Jena is designed to handle RDF and OWL ontologies. By this choice, we made possible (but not mandatory) for UFCL writers to reuse concepts from ontologies defined in RDF-S or OWL.
UFCL constructs were studied to know if it would be reasonable to directly compile them down to simple OWL; however UFCL and OWL have some important semantic mismatch. The main mismatch is between UFCL functionality facets and OWL classes. While in OWL, a given resource may have multiple types, it cannot "implement" twice the same type. In UFCL, the semantic of facets is such that a given resource may have multiple facets for a same functionality. It may implement the same functionality twice with different properties. Using directly OWL classes would have led to some aliasing problems where properties of different facets (but same functionality) would end being merged and confused.
To represent properly the semantic of our functionality facets using only triples, we have to reify our concepts. A service and each of its functionality facets are each represented by a RDF resource. Each facet's resource has a type (rdf:type like in RDF and OWL) and a set of properties. For example, in the UFCL statements from lines 1 to 6 (see previous section for code snippets), there would be one resource for the service itself. There would also be one resource for each functionality facet (two), each having a type property linking to the *Timer* resource and a *freq* property valued with respectively integers 2 and 1.
We described how functionality implementation descriptions using *isa* UFCL constructs are translated into triples in Jena's RDF database. Compilation of functionality correspondences that uses *a...isa* construct won't be detailed here as they can be reduced to a special case of factory com-

position.

5.2 Factory Descriptions Compilation

As factories may have free parameters like in the *Timer* example, we cannot have a purely forward inference system. A forward inference system applies all possible rules to produce new facts in its database and then allow one to query this database for its goal (production can also stop when the goal is reached). In the case of such open factories, forward rules would try to generate all possible *Timers*; however, there are an infinity of them and thus forward rules are not an option.

We opted for backward inference whose principle is to start from the expressed goal and use rules to rewrite this goal to existing facts. In our case, a query or a need for a given functionality facet would be rewritten using backward rules to a need for another facet (or more than one in the case of compositing factories) until existing facets fulfill these needs. In backward systems, an open factories like *Timer* factory is simply fulfilling any inferred need for a *Timer*.

Several inference systems such as prolog allow backward chaining. Jena is among them, however these inference systems are hard to master and it is difficult to have a fine control over how and where backward rules apply. Particularly, Jena imposes some restrictions on backward rules that are blocking in our case (a rule cannot produce both a new resource and multiple facts using it). We implemented a kind of backward chaining engine using forward rules: we reify the concept of "need" in the database and generate some need-rewriting rules. As rules are automatically generated, only the generation process is more complicated, nobody has to repetitively write more complex rules than in the forward case.

When one looks for a particular functionality facet, we can assert it as a needed facet coupled with its desired properties into the database. Each factory contributes to the rule-based system. For a factory requiring a list of functionalities, one rule will be generated for each successive required functionality and one rule for the final production of the factory product.

Back to the *Timer* factory described at lines 11 to 15, it does not depend on any functionality so it will produce only one rule. The only rule generated by our UFCL compiler is just a formal version of: "for any need for a *Timer* facet with a given *freq* property, create a new resource implementing a new virtual *Timer* facet with the right frequency". This makes any directly asserted or inferred need for a Timer to lead to the creation of a new functionality facet implemented by a virtual service. With this new virtual service and this new functionality facet, a description of how to instantiate this virtual service is inserted in the triples database. Instantiation information is basically composed of the factory grounding and some values for its free parameters (only *?pFreq* in this case). Instantiations information are read after the inference to know which factories to query and in which order to obtain the desired outcome.

The *Translator* composer described at lines 16 to 25 requires 2 existing functionality facets so the compiler generates 3 rules. First rule rewrites any need for a *Translator* from language A to language C to a need for a *Translator* from language A to anything. Second rule rewrites the previous need as soon as a Translator from language A to something else (say B) has been found; it generates a new need for a *Translator* from B to C. Last rule fulfills the original need when a *Translator* from B to C is found.

All these rules are made a little more complicated than it sounds. First, these rules must be designed not to fire recursively which would lead to infinite need rewriting. Second, all information eventually required to instantiate the service has to be transmitted through successive rules. In fact, a facet asserted in the triple database by the last factory rule is implemented by a virtual service from which information have to be extracted afterwards. Like in the case of *Timer* factory, this virtual implementer contains all information on how to get a real service instantiated by the factory: grounding, grounding message format and links to the *Translators* to compose.

UFCL lets one write some constraints in *having* section. In the example, only one constraint is given; however, the more constraints we have the narrower the search is. For example, when trying to compose two *Translators* to obtain one from A to C, we have an unbound intermediate language B. Adding the constraints that B (*?t1.to*) is different from both A and C, make the search faster. To maximize search efficiency and minimize what UFCL users have to write, we added a constraint inference mechanism. Constraint inference process is run before rule generation to infer implicit constraints due to transitivity and other properties (e.g. a = b and b $\neq$ c implies a $\neq$ c).

When a functionality facet is required, one has to gather all UFCL descriptions, compile them to triples and rules, assert his need in the triple database and run inference. One then has to read the triple database in order to find implementers and possibly choose among them. Selected plan may contain some "virtual" services that need to be instantiated by sending queries to factories. All this process is automated and user only needs to express what functionality he is looking for. By default, the plan requiring the less service instantiations is prefered. This "best plan" selection strategy will be extended by future works.

6 Experimentation and Example use case

In this section, we will present how UFCL can be used in the rearchituring of an existing system to make it more flexible and adaptable.

We propose to rearchitecture an existing system: the automatic cameraman from our team. This automatic cameraman is an important application in intelligent environments such as meeting rooms and amphitheaters. It uses cameras and microphones in the environment to detect the current activity based on different scenarios. Classical scenarios include conference talk, meetings and informal meetings or group discussions. Based on the currently perceived activity, the cameraman automatically produces a video montage out of the cameras and microphones by selecting the best point of view and the best microphones at each instant. The

result is an automatic recording of a meeting or conference talk. For a talk, during the presentation, recorded video is mainly showing the speaker, and its slides when they are changed or when the speaker is pointing at them. During question time, the cameraman records persons asking questions and the answers from the speaker. The cameraman produces a simple audio video file but could also annotate it with some links to presentation materials or, using SMIL, some annotations such as subtitles or speaker affiliation and name.

From an architectural point of view, this cameraman is composed of multiple distributed components but still has a static and monolithic architecture. The cameraman may be composed of elements such as cameras, microphones, visual tracking systems, speech detection systems, a slide change detector and a central reasoner. Activity recognition requires image and sound processing so it is mandatory to distribute processing. However, even if the cameraman is distributed, the central reasoner knows about all other elements. For instance it knows exactly how the recorder (movie maker) is started and it outputs exact commands to this recorder (e.g. record audio stream A3 and video stream V2). To generate these commands, the reasoner has to know about cameras and microphones which eventually makes it be coupled with all other components.

We rearchitectured the automatic cameraman in order to improve its flexibility and adaptability. Our approach is to separate concerns within the cameraman by splitting it hierarchically: we take an existing component, identify its main concerns and split it into an orchestrating part and some functional subparts. We transform the orchestrating part into a service or service factory that will require the abstract functionality concepts identified for the subparts. Each existing subpart is transformed into a service that is one implementation (but there could be others afterwards) of a subpart's functionality. When re-architecturing, we try to use factories as often as possible to improve design generalization and to maximize the number of services potentially available for a service consumer.

Splitting such an existing system is probably simpler using a top-down approach. Taking the cameraman from the top, it can be described as "a component that looks for the current interest space and record continuously this interest space". Based on this description, we split the system in a *Cameraman* that only orchestrates two other services: an *InterestSpaceEvaluator* and an *InterestSpaceRecorder*. The cameraman will only transfer interest space information (e.g. speaker's face location) produced by the *InterestSpaceEvaluator* to the *InterestSpaceRecorder*. This cameraman can be implemented as a factory that is parameterized by its working space (e.g. amphitheater A001) and could have the following UFCL description (grounding omitted):

```
composing
  a InterestSpaceEvaluator ?e
  a InterestSpaceRecorder  ?r
  a Locator ?le, ?lr
having
  ?le.what  = ?e
  ?lr.what  = ?r
  ?le.where = ?lr.where
gives a Cameraman
  with space = ?le.where
```

This code snippets shows more than what we described before as it uses *Locators*. Here again we separate concerns: we could have added a property to *InterestSpaceEvaluator* and *InterestSpaceRecorder* that tells us in which environment they are operating. This would have force any service exposing such a facet to locate itself. We externalized this location information which allows it to be brought by another service after startup: an automatic calibration process can easily provide such location information for cameras, microphones and other perceptual components. The part of existing cameraman that was determining what to record is separated and called *SituationModeler*. The *SituationModeler* must implement *InterestSpaceEvaluator* in order to be usable by the *Cameraman* factory. In the same way, existing media selection and recording part will be put in a *MovieMaker* service implementing *InterestSpaceRecorder*. Both of these functionality correspondences can be expressed using UFCL:

```
a SituationModeler
  isa InterestSpaceEvaluator
a MovieMaker isa InterestSpaceRecorder
```

We can split further both *SituationModeler* and *MovieMaker*. We present only the latter. Basically, we could keep a monolithic *MovieMaker* but we can distinguish two main concerns in it: given an interest space, choose the right camera and microphones; and, given this audio video stream, encode it and write it to a persistent storage. Here again we have two concerns leading to two subparts orchestrated by the *MovieMaker* defined by UFCL:

```
composing
  a AudioVideoFileWriter ?w
  a AudioVideoStreamSelector ?s
  a Locator ?l
having ?l.what = ?s
gives a MovieMaker
  with space = ?l.where
```

Splitting the *MovieMaker* allows us to easily substitute any of its subparts. Using a factory decreases deployment complexity and can be used to instantiate *MovieMaker* for other applications. We applied our recursive splitting process further. In particular our perception system can be cleanly spitted, making distinction between a camera, its calibration and the tracking systems using it. All this perception systems are used to eventually implement *InterestSpaceEvaluator* but also in the best camera and microphone choice made by the *AudioVideoStreamSelector*.

This system splitting task may look tedious but it basically consists in modeling software concepts present in the intelligent environment. In such environment, design and architecture are important. This modeling work may seem uninteresting but it is in fact mandatory if reusability and flexibility are targeted. Our language is designed to incite

people to better formalize their applications and thus leads to a better overall design. A better design means lower coupling, higher reusability and flexibility. Except substitutability and reuse of services, we can illustrate how separating concerns adds flexibility with the example of *Locators*. A factory can express that a given location is a sublocation of one another:

```
composing a Locator ?l
having ?l.where = A001
gives a Locator with what  = ?l.what
                with where = A-Wing
```

With this description, listing services located in the A-Wing will include anything declared to be in amphitheater A001. This refactoring of an existing automatic cameraman illustrates how our proposal makes it possible to apply a systematic separation of responsibilities by splitting system in low-coupled, substitutable and reusable components. Our usage of factories introduced in this article simplifies deployment of applications and makes them more dynamic. Eventually, using our language and factories brings a real service oriented architecture to the automatic cameraman and its underlying perception system.

7 Conclusion and Future Work

In this article, we proposed to augment classical semantic web services concepts with service factories. Service factories are services that exposes their ability to instantiate other services such as a *TimerFactory* or a *ComposedTranslatorFactory* instantiating respectively *Timers* at any desired frequency, and *Translators* by composing two existing *Translators*. We gave examples illustrating the benefits these factories bring to service oriented architectures. We also proposed the Usable developer-oriented Functionality Composition Language (UFCL), an accessible language to describe functionalities implemented by services and factories. We implemented a compiler from UFCL to a rule-based system; it allows us to reason about both available and instantiable services. We illustrated the advantages factories and our language bring with a concrete example: we rearchitectured an existing automatic cameraman that autonomously records seminars, talk or meetings.

Works on architecture and language design are difficult to evaluate properly. This article provides the reader only with a description and a mid-size proof-of-concept example but with no quantitative evaluation. One first direction for future work is to validate the language, by doing some user studies, and the compilation and reasoning system by evaluating its scalability to larger systems and its possible optimizations. This article does not handle protocol adaptation and sometimes our current system say "I would compose these if I knew how to do adaptation between their protocols". We can extend our work by adding an extensible and distributed protocol adaptation subsystem based on state of the art works. Dynamicity is one of our goals. For the moment, we only react to dynamic additions and removals of services and associated descriptions by computing a new composition. We plan to add capabilities to reevaluate only subparts of the composition in order to replace services that disappeared or to make better choices among new ones.

References

[1] OSGi Alliance: http://www.osgi.org/.

[2] Jena Semantic Web Framework: http://jena.sourceforge.net/resources.html.

[3] André Bottaro, Anne Gérodolle and Philippe Lalanda. Pervasive Spontaneous Composition. In *SIPE*, 2006.

[4] Steffen Balzer, Thorsten Liebig, and Matthias Wagner. Pitfalls of OWL-S – A practical Semantic Web Use Case. In *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC004)*, 2004.

[5] D. Berardi, D. Calvanese, G. DeGiacomo, M. Lenzerini, and M. Mecella. Eservice composition by description logics based reasoning, 2003.

[6] H. Cervantes and R. S. Hall. Automating service dependency management in a service-oriented component model. In *Proceedings of the 6th Workshop, Component Based Software Engineering*, 2003.

[7] Humberto Cervantes and Richard S. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. *icse*.

[8] Diego Lopez de Ipina and Sai Lai Lo. Locale: A location-aware lifecycle environment for ubiquitous computing. In *ICOIN*, 2001.

[9] Rémi Emonet, Dominique Vaufreydaz, Patrick Reignier, and Julien Letessier. O3miscid: an object oriented opensource middleware for service connection, introspection and discovery. In *SIPE*, 2006.

[10] Rania Khalaf, Nirmal Mukhi, and Sanjiva Weerawarana. Service-oriented composition in bpel4ws. In *WWW (Alternate Paper Tracks)*, 2003.

[11] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara. Bringing semantics to web services: The owl-s approach, 2004.

[12] Jörg Nitzsche, Tammo van Lessen, Dimka Karastoyanova, and Frank Leymann. BPEL for semantic web services (BPEL4SWS). In *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*, 2007.

[13] Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, 1999.

[14] Z. Wu, A. Ranabahu, K. Gomadam, A. P. Sheth, and J. A. Miller. Automatic composition of semantic web services using process and data mediation. Technical report, 2007.