BIONETS

WP 3.2 – AUTONOMIC SERVICE LIFE-CYCLE AND SERVICE ECOSYSTEMS

Advanced Service Life-Cycle and Integration

Deliverable 3.2.4

Reference:	BIONETS/VTT/WP3.2/v1.0
Category:	Deliverable
Editor:	Janne Lahti (VTT)
Co-authors:	Francoise Baude (INRIA), Laura Ferrari (TI), Ludovic Henrio (INRIA), Ville Könönen (VTT), Daniele Miorandi (CN), Heiko Pfeffer (TUB).
Verification:	Paolo Dini (LSE), Daniel Schreckling (HITeC)
Date:	July 30, 2008
Status:	Final
Availability:	Restricted

Executive Summary

In this document we present the status of the work in several topics researched in the BIONETS WP3.2. We illustrate the BIONETS service evolution by presenting it at the composition level as well as providing a framework for the cooperative evolution of services. This deliverable also presents the refinements of self-management concepts, containing the specification of semantic injection in the BIONETS service architecture, semantic service specification, and a summary of the achievements in migration support for evolutionary services.

We present a framework for the cooperative evolution of services. We explain the principles of the approach for atomic stateless services, and then show how this approach can be extended to stateful services or service assemblies.

We introduce the modelling of Service Cells (SCs) and Service Individuals (SIs) to demonstrate SI evolution. We aim at transforming already existing service individuals in order to create functionally equivalent SIs that may differ in their non-functional properties. This procedure is also advantageous since the complex semantic service descriptions can be overtaken; thus, there is no need to generate new descriptions.

For the semantic injection in the BIONETS service architecture, we consider the research trends in distributed ontologies and semantic reasoning, and see how this can be applied to the BIONETS service architecture, i.e., Service Cell and Individual. The main ideas developed are to characterize the BIONETS Service Framework with a lightweight and distributed semantic annotation and able to deal with distributed semantic reasoning.

For the semantic service specification we propose an extension and adaptation of the OWL-S ontology that is suitable for the BIONETS Service Description. The idea is to start from OWL-S, which guarantees a good base and a lot of background and specification activities (see W3C fonts and Semantic Web), and takes in consideration the most suitable aspects for BIONETS. Mainly the idea is to develop and study a possible incremental approach in service description and semantic annotation of services.

Finally, for the migration support for evolutionary services, we refine the notion of service migration within the context of atomic service cells. We clarify the definition of service migration, present a high-level scenario of service mobility in the BIONETS environment, and analyze the requirements for different components and functionalities. We also present research done in the specific topics related to the service migration, like decision-making mechanisms.

Document History

Version History

Version	***Status***	***Date***	***Author(s)***
0.1	*ToC*	*2008-05-08*	*Janne Lahti*
0.2	*revised ToC*	*2008-05-15*	*Janne Lahti*
0.3	*revised ToC*	*2008-05-27*	*Janne Lahti*
0.4	*First contribution from VTT, CN and TI.*	*2008-06-16*	*Janne Lahti, Daniele Miorandi, Laura Ferrari*
0.5	*First contribution from INRIA and TUB. And update to 3.2*	*2008-06-25*	*Janne Lahti, Ludovic Henrio, Heiko Pfeffer*
0.6	*Update to 1.2*	*2008-06-27*	*Ludovic Henrio*
0.7	*Small editing*	*2008-07-01*	*Janne Lahti*
0.8	*Ex.Summary +Intro and update to 3.2*	*2008-07-01*	*Janne Lahti, Ville Könönen*
0.9	*Updates to Intro, Ex.Summary and section 3*	*2008-07-03*	*Janne Lahti, Ludovic Henrio, Heiko Pfeffer, Laura Ferrari, Daniele Miorandi*
0.10	*Ready for review*	*2008-07-04*	*Janne Lahti*
0.11	*Reviewed*	*2008-07-21*	*Daniel Schreckling, Paolo Dini*
0.12	*Changes accepted*	*2008-07-23*	*Laura Ferrari*
0.13	*Changes accepted*	*2008-07-23*	*Janne Lahti*
0.14	*Changes accepted*	*2008-07-28*	*Ludovic Henrio, Daniele Miorandi*
0.15	*Changes accepted*	*2008-07-29*	*Heiko Pfeffer*
1.0	*FINAL*	*2008-07-29*	*Janne Lahti*

Summary of Changes

Version	***Section(s)***	***Synopsis of Change***
0.1	*All*	*ToC proposal*
0.2	*ToC*	*ToC new version*
0.3	*ToC*	*ToC new version*
0.4	*2.1, 3.1, 3.2*	*First contributions*
0.5	*2.1,2.2,3.2*	*First contributions + updates to 3.2*
0.6	*2.1*	*Updated content*
0.7	*All*	*Small editing*
0.8	*All*	*Small editing + Ex. Summary + Intro*

0.9	*All*	*Small editing*
0.10	*All*	*Final editing*
0.11	*All*	*Reviews*
0.12	*3.1, 3.2*	*Accommodating reviewer's comments*
0.13	*Ex.Summ., Intro, 3.3*	*Accommodating reviewer's comments*
0.14	*2.1*	*Accommodating reviewer's comments*
0.15	*2.2*	*Accommodating reviewer's comments*
1.0	*All*	*Closing*

Contents

1. Introduction

In this document we present the status of the work in several topics researched in the BIONETS WP3.2. We illustrate the BIONETS service evolution by presenting it at the composition level as well as providing a framework for the cooperative evolution of services. This deliverable also presents the refinements of self-management concepts, containing the specification of semantic injection in the BIONETS service architecture, semantic service specification, and a summary of the achievements in migration support for evolutionary services.

In 2.1, we present an approach for the autonomous evolution of services, where best services are selected and genetic operators applied to their genotype. This operation creates a new service. We detail the approach for atomic stateless services in the first instance, and then show the possible extensions to stateful and hierarchical services. While this approach focused on the evolution of services themselves, it only allows basic evolution of service dependencies. In 2.2 we show how complex evolution of interconnections can be applied to basic services.

In 3.1, we clarify the semantic injection in the BIONETS service architecture; we elaborate, i.e., the lightweight semantics and distributed reasoning for the BIONETS Service Architecture, the semantic support for BIONETS evolutionary service lifecycle, and the automatic service composition based on semantics.

In the BIONETS project there is no intention to develop a new semantic approach; it is not a BIONETS objective as many other projects and organizations are working on it, but it would be useful to consider the research trends in distributed ontologies and semantic reasoning, and see how this can be applied to the BIONETS service architecture, i.e. Service Cell and Individual. The main ideas developed are to characterize the BIONETS Service Framework with a lightweight and distributed semantic annotation, while the Mediator should be able to deal with semantic and distributed semantic reasoning. Semantics needs to evolve: semantic adaptation, merging of semantic concepts and relationships, and semantic matching can occur for the BIONETS Service Cell and Individual.

In 3.2, we propose an adaptation of the OWL-S ontology for the BIONETS service description. We start from OWL-S, which guarantees a good base and a lot of background and specification activities, and take in consideration the most suitable aspects for BIONETS. Mainly, the idea is to develop and study a possible incremental approach in service description and semantic annotation of services. Not all the standard and specification aspects can be covered and are needed in BIONETS. We will sketch a possible approach and service description covering some of the aspects that seem more relevant for BIONETS, and specifically we take in consideration an incremental approach.

In 3.3, we will first clarify the definition of service migration. Then we present a high-level scenario of service mobility in the BIONETS environment and analyze the requirements for different components and functionalities. We also present research done in the specific topics related to service migration, like decision-making mechanisms.

In 4, we outline the conclusion for the deliverable and presents further steps for the research topics covered.

2. Evolution of Services

2.1 A Framework for the Cooperative Evolution of Services

In this section, we will present a framework for the cooperative evolution of services. The process is called “cooperative” as it is based on the exchange of information among different nodes in the system on the performance of “similar” services (i.e., services with the same semantic description [1]). The framework is drawn building on for the experience gained from the experiments performed on the evolution of epidemic-style relaying protocol [20], [21]. This section first explains the principles of the approach for atomic stateless services, and then shows how this approach can be extended to stateful services or service assemblies.

2.1.1 Basic Framework: Atomic Stateless Services

For the sake of simplicity, we consider a situation in which different instances of one single service (one single species) are present in the system.
We start from the following assumptions:

- each service can be expressed by a “genotype” which can be processed by the node to obtain the corresponding “phenotype” (executable instance of the service); [1]
- each node employs suitable mechanisms for estimating the fitness of the running service instance;
- each node maintains a pool of service genotypes, together with the corresponding fitness values;
- one of the genotypes is selected for execution.

The system model is depicted in Figure 1.

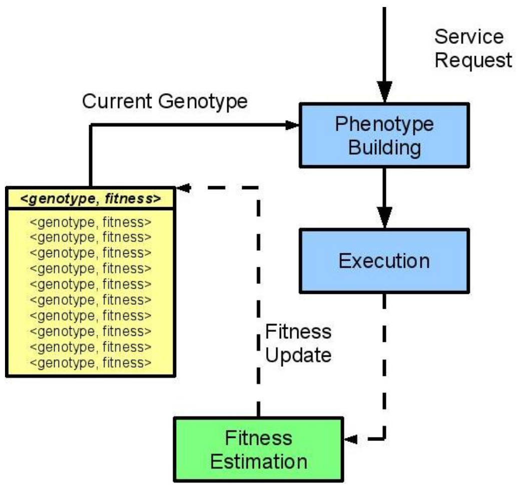


Figure 1: Cooperative evolution of atomic services: system model

When two nodes meet, they can exchange information on the services they are currently running and the respective performance. The data is exchanged in the form <genotype, fitness>, and the information received is added to each node's pool, as plotted in Figure 2. In order to limit the amount of data to be exchanged, only information on the genotype in use is exchanged.

In this way, nodes can gather information on the performance (expressed by the fitness level) of similar services. This defines a sort of (indirect) context sensing process. Clearly, the whole framework is built under the assumption that the different fitness levels, as estimated by different nodes, are consistent. In general, however, different nodes may be operating in

[1] The genotype could correspond, depending on the specific system implementation, to, e.g., a set of parameters used to configure/instantiate a service, a description of an algorithm used within a service etc. In general, service genotypes should be light enough to enable their transmission during short contact opportunities.

different contexts, resulting therefore in a sort of "noisy" fitness estimation (the noise corresponding to the impact of context-specific factors, which are not referable to the actual service performance). The next step is to devise suitable mechanisms for making use of the information gathered in such a way to enhance the current running service. Such evolutionary steps should be based on the use of the information contained in the pool only. They can take place either periodically or when the number of items in the genotype pool has reached a sufficiently large value. In any case, the evolutionary process has to happen on a time-scale much larger than the one associated with the service run. At the end of the evolutionary step, a new genotype is chosen to be executed. The pool is emptied, and the whole process restarts. In general, it is not mandatory to empty the pool; such a choice comes from experiments performed in SP1 with protocol evolution (see, e.g., [2]), where it has been discovered that such a choice enhances the performance of the system in terms of its ability to track varying environmental conditions.

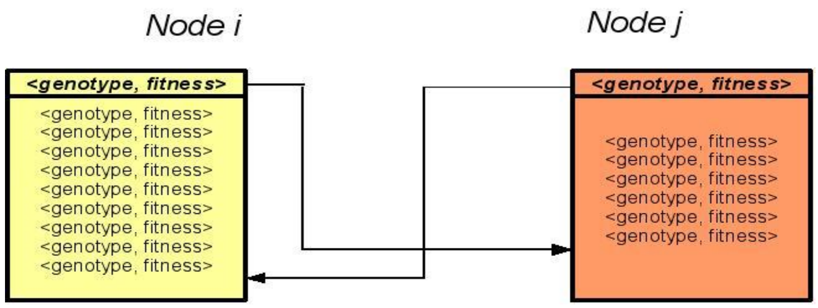


Figure 2: Cooperative evolution of atomic services: genotype exchange

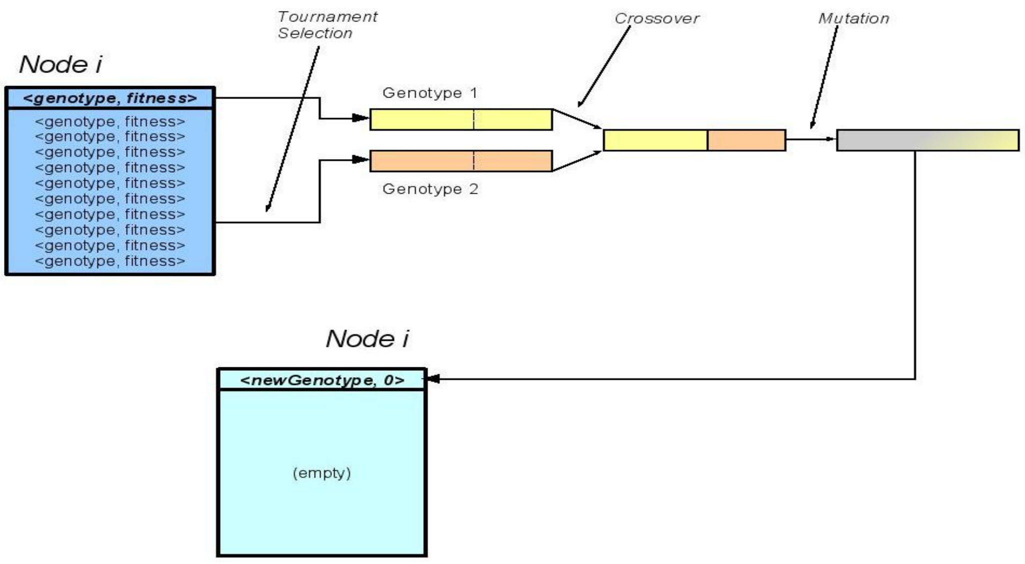


Figure 3: Cooperative evolution of atomic services: a possible GA/GP-based implementation of the evolutionary step

In general, the specific mechanisms to be employed depend heavily on the specific genotype representation used, and can leverage the machinery developed in the Evolutionary Computation field. One of the possible mechanisms to be employed relies on the use of tools and techniques taken from the field of Genetic Algorithms (GAs)/Genetic Programming (GP) field [22]. In such a case, two genotypes are selected from the pool according to some fitness-dependent mechanism. Based on the experiments performed on epidemic-style data dissemination [20], [21], we recommend the use of the running service as one of the two genotypes, the other one being drawn according to a tournament selection procedure. (This can be interpreted as a form of elitism in GA/GP.) Cross-over is then applied (with a given probability), and one of the two resulting genotypes chosen at random. With a given probability, mutation is applied to the resulting genotype. In some cases, constraints on the form of the genotype may be present (as not all possible genotypes may lead to consistent and/or valid phenotypes). In such cases, a consistency check is applied, in order to make sure the resulting genotype is a valid one (otherwise, the process is repeated until the condition is met). In order to clarify, let us consider the case in which the genotype corresponds to a real-valued parameter used for instantiating the service, and assume that such parameter should take value in [0,1]. As customary in GA for real-valued variables, mutation is performed by adding a quantity generated according to a Normal distribution with zero mean and appropriate variance. In this case it may happen that the resulting value falls outside the [0,1] range: the process is therefore repeated until a valid one is obtained.

Such genotype is then used to build the phenotype and executed. Correspondingly, the pool is emptied and the gathering/exchange of information is started again. The fitness value of the running phenotype is set to zero, until a reasonable estimate of its performance can be obtained. The whole process is illustrated in Figure 3.

It is worth remarking that, as the evolutionary step is performed periodically, at a given time instant there may be genotypes in the pool whose estimated fitness value is larger than that of the genotype currently in use.

2.1.2 Extension to Stateful Services

Considering stateful services, state-transfer mechanisms could be envisaged: such mechanisms are used in software engineering to transfer the state between two known components of the running system: when one component is replaced by another one, the state is transferred using a state transfer function that is application- dependent (and even instance-dependent). Like the mechanisms above, the meaning of a state-transfer function heavily relies on the genotype (and probably does not exist for most of the genotypes), but supposing such a function can be provided by the service itself, it allows the transfer of the preceding reasoning to stateful services. The transfer function mainly depends on the service, but also on the changes in the genotype.

Such state transfer somehow implies the replacement of a service by another one, or the creation of a service from a parent, giving its state. Stateful services that cannot be given an original service to take their state from must evolve as stateless ones: with an initial state given to them. Transferring a state from a whole population to a single new service would be very difficult to define.

Then, the state transfer occurs at the last step of evolution: when a service is created from a phenotype, it replaces another service, and takes its (transferred) state.

2.1.3 Extension to Composite Services

Let us consider now a composition of services, i.e. a composite service. As in the case of an atomic service, we focus on the simple case where several instances of the same service are currently running on the distributed set of nodes. Here services are considered as identical if they fulfil the same role and are sufficiently similar. The methodology suggested here is particularly adapted to identical compositions built from different individual services. We will also show hoe to adapt it to the same service composed in different ways.

In the context of service composition, we claim that the phenotype can be considered to be the deployment plan that is the description of the effective architecture of the service composition. (i.e. the description of an organism). Note that it is from such a plan, that a specific service individual is effectively created. As the organism is implicitly driven by its genotype, a genotype should be sufficient to generate such an architectural description; we call it the *abstract* architectural description. An effective architectural description describes a service as a composition of services interconnected by some bindings. Each sub-service has then also its phenotype. At the leaves of the genotype hierarchy are the genotype for atomic services described above. Selection, crossover of genotypes, evaluation of fitness and evolution can be performed as described for the atomic services, at each level of the hierarchy. At each level the services involved in the composition evolve in a non-hierarchical manner, but this evolution is applied at each level of the hierarchy, leading somehow to hierarchical evolution.

Combining genotypes, operating on them with genetic operators, and selecting according to fitness can then be performed in the same way for composite services as for atomic ones.

Figure 4 illustrate the hierarchical composition of genotypes, the composed genotype, simply store the information that services S1 and S2 are to be composed, and also remembers the interconnections between S1, S2, and the composite service. The composition is hierarchical because S1 and S2 are themselves services, either individual ones, or similar to the composite service. Evolution occurs at each level of the composition in a similar way.

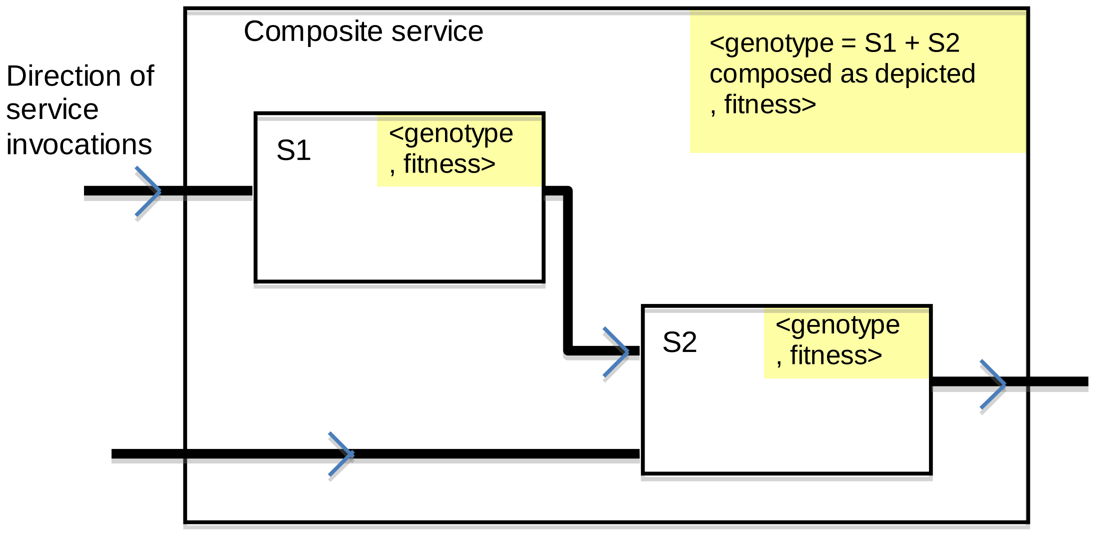

Composite service
<genotype = S1 + S2 composed as depicted , fitness>
Direction of service invocations
S1
<genotype , fitness>
S2
<genotype , fitness>

Figure 4: Hierarchical composition of genotypes

The description of hierarchical composition of fitness is very specific to the definition of fitness and cannot be composed as easily as the genotypes. For example, composition is easy to compute if fitness is the resource consumption, as it is roughly the sum of the consumption of each sub-service. In general (performance, satisfaction of user requirements) composing fitness is much more complex.

If all running instances of the service are composed in the same way (they follow the plan), it is easy to perform crossover and mutations: members of the composition are modified by those operations, and interconnections stay the same (i.e. evolution may end up in deciding to replace an inner service S2 by some equivalent but more fit one, noted S’2). Else, the situation is much more complex, but one could always crossover and mutate the atomic service members of the composition independently, and try to automatically produce a correct interconnection (relying on service type and semantic description). If such an interconnection cannot be automatically found, then the crossover + mutation step is performed again until a feasible composition is found.

2.1.4 Service Instantiation vs. Service Reconfiguration

For both atomic and composite services, one has to choose, when a new phenotype has been obtained, how to get a service from it. More precisely, we have the choice between picking an already running service and making it evolve, or creating a new one.

For atomic services, the choice depends on the nature of the service and on its evolution. For simple evolutions where the global specification of the service stays the same (e.g. change in parameters, or in the way services are composed without impacting the way services can be addressed), it is possible to make the service evolve, for example when a parameter has to be updated. But, in general it is necessary to create a new instance from the new genotype.

For composite services, the choice depends on the service framework: if the framework supports reconfiguration of the service composition (i.e. changes in binding, suppression and addition of services into an existing one), then services can be reused; else a new service has to be instantiated. The GCM implementation [3] is a component framework (proposed in the context of the project to act as a BIONETS service framework) supporting complete reconfiguration of both the functional and the non-functional parts of the service composition.

In Section 2.1 we have shown how cooperative evolution can be extended to composite services, including complex ones, provided the interconnections between evolving services remain the same. In case interconnections have to change, we rely on some compatibility relation (e.g. type compatibility, possibly relying on behavioural types), expecting a feasible composition can be found. Section 2.2 on the contrary starts from more basic services but show how, in that case, bindings themselves could be subject to evolution.

2.2 Service Evolution at Composition Level

In this section we introduce the modelling of Service Cells (SCs) and Service Individuals (SIs) to demonstrate SI evolution. The SIs are represented by a set of service blueprints and two control graphs, a workflow graph and a dataflow graph, as introduced in deliverable D3.2.3 [4].

We aim at transforming already existing service individuals in order to create functionally equivalent SIs that may differ in their non-functional properties. This proceeding is also advantageous since the complex semantic service descriptions can be overtaken; thus, there is no need to generate new descriptions. Starting with an initial set of SIs, we transform existing SIs through the application of genetic operators in order to create new SIs; finally, we compare the generated compositions with regard to their functional similarity by simulating the SIs execution with random inputs and comparing their respective outputs. Detailed simulation results can then be found in deliverable D3.4.1 addressing the outcome of service probes.

In section 2.2.1, we outline the representation of variables and their respective domains. Section 2.2.2 then discusses the requirements of the service model and the corresponding primitive operations that services can perform on variables defined in the previous section. Section 2.2.3 discusses the initialization and parameterization of simulation runs. Here, a SI derived through the application of genetic operators to an already existing SI is executed with random inputs in order to compare its functional behavior to other SIs. Section 2.2.4 introduces the genetic operators that are used to modify the SIs during the simulation runs. The evaluation of the degree of similarity between two SIs is finally discussed in section 2.2.5.

2.2.1 Data Domains

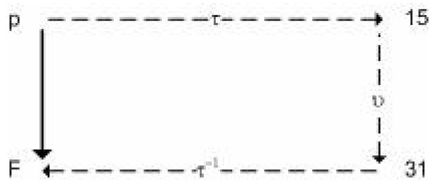
Each variable can be defined by two characteristics: its domain space, i.e. the set of values it can take, and its current value. In order to abstract from the single data types present in current computing systems, we reduce variables to these two identified characteristics and thereby derive two valuable simulation parameters.

Each variable has a unique *domain space*. A domain space is given by a finite set of integers arranged as a *cyclic group*[2] Thus, a variable x may have the domain A with $A = \mathbb{Z}_n = \{0, \dots, n-1\}$. Therefore, each variable can be modelled with its essential characteristics, i.e. a domain space and a current value, while reducing the value of variables to numbers in $\mathbb{Z}$. For instance, $B = \mathbb{Z}_2 = \{0, 1\}$ may represent the data type of Booleans while

[2] 1 A *cyclic group* is a group that can be generated by a single element *g*, the so-called group generator, such that every element of the group is a power of *g* (with the notion of multiplication) or a multiple of *g* (with the notion of addition).

$C = \mathbb{Z}_{52} = \{a, b, \ldots, z, A, B, \ldots, Z\}$ may denote the domain of characters. We write $x: A$ to indicate that the variable x has the domain A.

Thus, we identify every finite domain by a cyclic group containing an enumeration of the elements of the respective domain. This (bijective) mapping τ allows us to modify all variables with the same operators, independent from the single data types; these operators are discussed below within section 2.2.2. For instance, the domain of characters is mapped to $\mathbb{Z}_{52}$; then, unified operations on this cyclic group can be performed to transform the value of the variable. The result of this transformation is than mapped back to the domain of characters; an example of this mapping is illustrated in Figure 5.



$$\begin{tikzcd}
p \arrow[r, dashed, "\tau"] \arrow[d] & 15 \arrow[d, dashed, "\upsilon"] \\
F & 31 \arrow[l, dashed, "\tau^{-1}"]
\end{tikzcd}$$

Figure 5: Transforming variables represented by cyclic groups

For a simulation setting, we can thus define a finite set of different data types together with their respective domains. These two parameters, i.e. the number of domains and the size of the single domains, can also be used as simulation parameters, influencing considerably the success of the SI evolution procedure. In the following section, we discuss the requirements for single SCs and derive atomic operations they can perform on the variables introduced within this section.

2.2.2 Service Capabilities

Service Cells are considered as atomic executions of application logic. Therefore, a single service cell should not be decomposable into multiple functionalities. Moreover, we want operations to remain as simple as possible to ease the simulation runs while ensuring that the respective functions operate on the whole domain space of the given variables.

The consideration of domain peaks, i.e. the fact that variables more often take values from a specific region of the domain space than from another, is considered as future work.

We define the following elemental service functionalities, as depicted in Figure 6.

add(x) – A variable k of the domain A is augmented by the value x, $0 \le x \le n$ where n is the domain size of A. Since the resulting value can exceed the domain of A, the result is derived by modulo n. Thus, a domain is always considered as a cyclic group with addition[1].

split – A single input k is copied to multiple output ports.

merge – Multiple inputs are combined to a single output; therefore, the respective inputs are added in modulo fashion. Note that the input variables have to be of the same domain A.

cast(B) – Casts the input variable to the respective domain, here B.

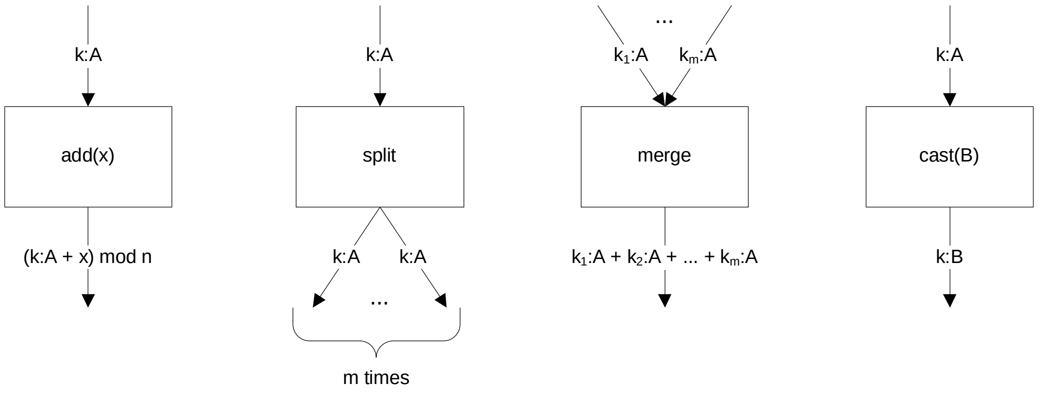


Figure 6: Four basic Operations performed by Service Cells

These elementary operations can be combined in order to achieve higher-level functionality. For instance, a variable can be modified and cast as depicted in Figure 7.

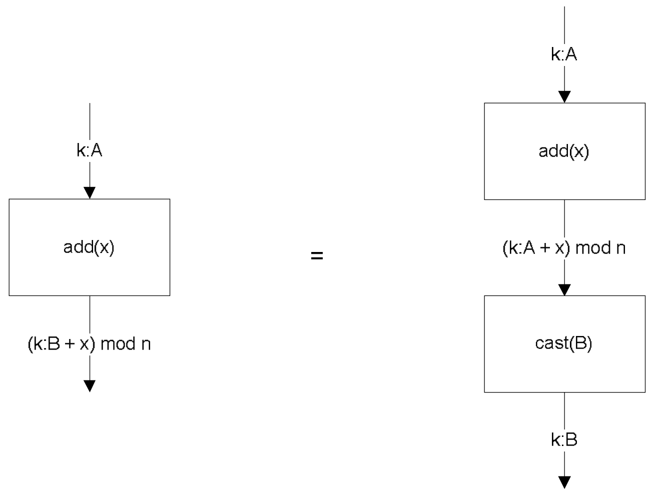


Figure 7: Service Individual for casting and modifying a variable k

In a similar way, the simple split operation can be used to map one single input to multiple outputs differing in their value. The corresponding composition of services is illustrated in Figure 8.

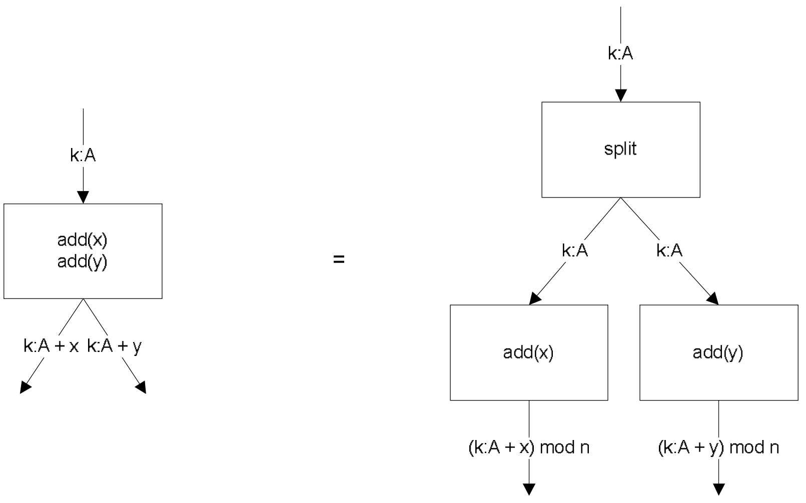


Figure 8: Generating multiple different outputs from one input

Within deliverable D3.2.3 [4], we introduced a service model featuring I/O descriptions extended by semantically described gains, i.e. outputs with importance beyond the execution of a single service. The service model is again depicted in Figure 9.

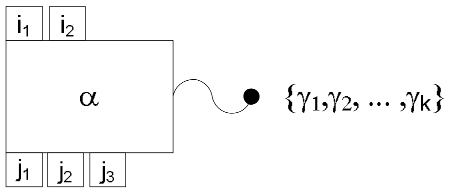


Figure 9: Service model (introduced in deliverable D3.2.3) featuring gains

Each service is annotated by a set of gains $\{\gamma_1, \ldots, \gamma_k\}$ that may be generated during service execution. However, it is not previously known which subset of gains will be generated, since the generation of gains can depend on input values and the availability of other gains. Within the simulation, we handle gains as follows. For every service, a subset of values from the input variables' domains is selected, indicating when the gain is generated. For instance, in case a service has one input of the domain $G = \mathbb{Z}_{100} = \{0, \dots, 99\}$, it may be defined that the gain γ_i is generated if the variable is between 30 and 40. These intervals for the generation of gains are randomly assigned to SCs when they are initially added to a SI. Every time a gain is generated, it is pushed into the gain queue as introduced in deliverable D3.2.3 [4].

2.2.3 Simulation Settings

The simulation is divided into three elementary parts:

1. Generation of initial SIs
2. Transformation of SIs
3. Evaluation of SIs

Thus, a selected SI of an initial set of SIs is transformed through the application of genetic operators and then evaluated with regard to its functional similarity compared to the other SIs. The parameters for the simulation are summarized in Table 1 and discussed in greater detail within the remainder of this section.

Type	Parameter	Description
int	**numberOfDomains**	Specifies the number of different domains for a given setting, i.e. the number of simulated data types.
int[1..numberofDomains]	**domainSize**	Specifies the domain size for each domain. A domain size of n corresponds to a domain $\mathbb{Z}_n = \{0, \dots, n-1\}$.
int	**initComp**	Specifies the number of service individuals that should be initially generated.
int[add, split, merge, cast]	**serviceProb**	Defines the probability of the occurrence of the elementary service cells within the service individuals. For instance (0.5, 0.3, 0.1, 0.1) defines that a service cell within a SI is with 50% probability an add operation, with 30% probability an split operation, and with 10% probability an merge or cast operation, respectively.
int	**avServices**	Defines the average number of services in one service individual
int[low, high]	**limServices**	Defines the lowest and the highest number of services allowed within one service individual.
int	**opRuns**	Specifies the number of SI modifications, i.e. how often genetic operators are applied to SIs.
float[mutation, crossover]	**opProb**	Defines the probability for the application of the mutation and the crossover operation, respectively.
int	**parallelComp**	Defines how many SIs are developed simultaneously at most.
int	**simRun**	Specifies the number of evaluation runs.

Table 1: Simulation Parameters

Within the first phase, the number of domains is defined as well as their respective domain size (*numberOfDomains*, *domainSize*). Moreover, the initial set of available SIs is defined (*initComp*, *serviceProb*, *avServices*, *limServices*). The second phase contains the modification of the already existing SIs by genetic operators. Here, we can define or specify how often genetic operators should be applied (*opRuns*), with which probability the mutation or crossover operator is applied (*opProb*), and how many SIs are transformed in parallel (*parallelComp*). The last parameter enables the restriction to fewer SIs that are –in turn– modified more often. For instance, by setting *opRuns* to 10 and *parallelComp* to 3, the 10 genetic operators are applied to three different SIs at most. Within the third phase, the newly generated SIs are evaluated. Therefore, each new SI derived through the application of at least one genetic operator is compared to all initial SIs. The evaluation is based on the execution of SIs with randomized inputs. Generating a random input for all SIs, executing the SIs and deriving the respective outputs is considered as one simulation run. The number of simulation runs that should be performed before the SIs are evaluated can be defined by the parameter *simRun*.

In the next section we introduce the genetic operators that are used to transform the SIs. In the last subsection, we then discuss how SI are evaluated after a finite set of simulation runs.

2.2.4 Genetic Operators

In this section we discuss the modification of service compositions by genetic operators. Since service compositions are described as graphs, the modification of a service composition is realized as a structural transformation on its graph representation. The aim of the transformation is to vary the service composition structure in order to:

1. Replace service types by more accurate ones, with respect to the addressed problem
2. Substitute services referred to in the workflow, when no corresponding service instance is available in the current computing environment

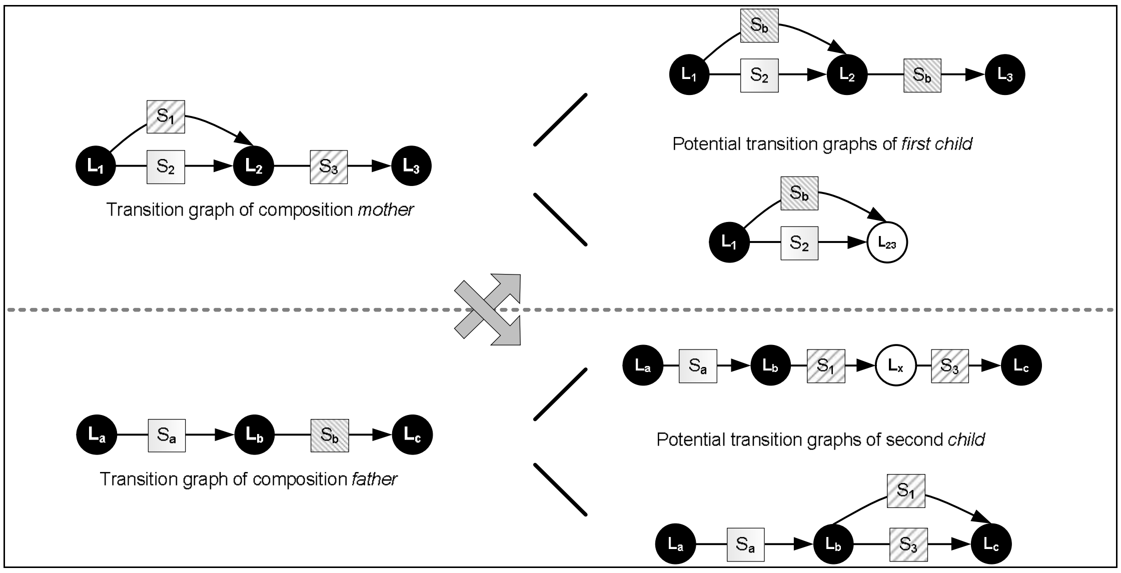
For these purposes we started developing two genetic operators to be applied to service composition graphs. The first one is a *crossover* operator, which involves at least two service compositions. The second one is a simple *mutation* operator to be applied to single service compositions.

The crossover operator takes two service composition graphs, in the following referred to as *mother* and *father*. The service compositions should be known to perform semantically similar tasks to forward the usefulness of the result. The operator randomly selects elements from both graphs and exchanges them. In practice, the algorithm for the crossover operator is composed of two steps. In the first step service references (given as transitions) from both compositions are selected and swapped, in the second step the dataflow is repaired.

For swapping transitions between the workflows a random number of transitions from both graphs is selected. These two sets of selected transitions are cut out of the graphs they originated from. The set of selected transitions (i.e. the referenced services) from the composition graph *mother* is then pasted into *father* and the set of selected transitions from *father* is pasted into *mother*. For this pasting the locations of the cut out transitions are used in order to recreate a consistent workflow. The transitions pasted into the workflow are at random arranged either sequentially or in parallel. Figure illustrates this exchange of transitions. If there are fewer transitions to paste available than transitions between locations that need to be fixed, transitions are randomly cloned or two locations combined.

After swapping the transition between mother and father, the overall dataflow of the resulting graphs is likely to be broken. Thus, a rearrangement of the port mappings between the old and the new transitions is required. For this purpose, the input ports of all transitions are checked. If they are not assigned to an output port of a transition precedent in the workflow, a back-tracking search in the workflow is performed to find an output port (including the overall composition inputs) with the same type as the input port to be bound. Instead of taking the first match, all candidates are collected and afterwards one is chosen at random. If input ports remain unbound anyway, the entire service composition is discarded.

The *mutation* operator removes information from the service composition or includes new information. In practice, the application of the operator to a service composition may either change dataflow or workflow and dataflow. In the first case, a random number of connections from service output ports to service input ports are broken up and the input ports allocated with correctly typed constants. In the second case, transitions are removed from the workflow or new transitions added. The removal of transitions may cause an inconsistency in the workflow and require the combination of the two locations spanned by the removed transition. If the workflow is consistent again, the dataflow is repaired in the same way as the child originating from a crossover operation described above. If the dataflow cannot be recovered, the mutated service composition is discarded.



Transition graph of composition *mother*
Potential transition graphs of *first child*
Transition graph of composition *father*
Potential transition graphs of second *child*

Figure 10: Genetic Operators: Crossover

2.2.5 Evaluating Service Individual Similarities

As introduced in deliverable D3.2.3[3], the workflow graph of a SI contains two pseudo-states, one initial state capturing the user input and a final state accepting all outputs of the SI. Therefore, the inputs as well as the outputs can be considered as ordered, since they are assigned to a unique port of the pseudo-initial state or final state, respectively.
We express inputs, outputs and gains of a SI with three vectors, i.e. an input vector $\vec{p} = (p_1, \dots, p_m)$, an output vector $\vec{q} = (q_1, \dots, q_n)$ and a gain vector $\vec{\lambda} = (\lambda_1, \dots, \lambda_k)$. We then calcualte the level of similarity by computing the distances of the vectors the SIs create.

2.2.5.1 Comparing Outputs and Gains

Within the simulation, we only compare SIs featuring matching input vectors; here, two vectors *match* iff they have the same dimension and every dimension is bound to the same domain space. E.g. $\overrightarrow{p_A} = (5{:}\mathbb{Z}_{10}, 0{:}\mathbb{Z}_2, 42{:}\mathbb{Z}_{100})$ and $\overrightarrow{p_B} = (7{:}\mathbb{Z}_{10}, 1{:}\mathbb{Z}_2, 77{:}\mathbb{Z}_{100})$ are matching vectors. $\overrightarrow{p_C} = (2{:}\mathbb{Z}_{10}, 1{:}\mathbb{Z}_2)$ does not match $\overrightarrow{p_A}$ or $\overrightarrow{p_B}$ since it has a different dimension, $\overrightarrow{p_D} = (3{:}\mathbb{Z}_{22}, 0{:}\mathbb{Z}_2, 33{:}\mathbb{Z}_{100})$ does not match $\overrightarrow{p_A}$ or $\overrightarrow{p_B}$ since the domain of the first dimension is different.

Beside the output vectors, SIs generate a finite set of gains. Since we are interested in the evaluation of the functional similarity between two SIs, we only regard gains that have been generated during the execution of a SI instead of considering all gains that can be potentially generated. For instance, a SI *A* generating the gains $\overrightarrow{\lambda_A} = (\lambda_1, \lambda_2)$ and a SI *B* generating $\overrightarrow{\lambda_B} = (\lambda_2, \lambda_3, \lambda_4)$ have both led to the generation of the gain λ_2 while they differ in the other gains they created. Therefore, the gain vector $\vec{\lambda}$ of the combination of the two SIs contains *n* elements, where *n* is the number of different gains generated by the SI itself and the SI it is compared to. Referring to the example above, the gain vector for the SI *A* and *B* would be $\overrightarrow{\lambda_{A,B}} = (\lambda_1, \lambda_2, \lambda_3, \lambda_4)$. Every gain has a Boolean domain space, i.e. $\lambda_i \in \mathbb{Z}_2 \; \forall i$, where the value is set to 1 if the gain has been generated during execution, otherwise to zero. Since the gain vectors of the SIs that should be compared are defined in a way that they have the same length, they can be combined with the respective output vectors, as explained below.

2.2.5.2 Evaluating the distance index between two SIs

Thus, we append the gain vector to the output vector in order to ease the comparison; the resulting vector is referred to as $\overrightarrow{q^\circ\lambda} = (q_1, \dots, q_n, \lambda_1, \dots, \lambda_k)$ in the following.

Every time a SI is executed with a specific input $\vec{p}$, a verctor $\overrightarrow{q^\circ\lambda}$ is generated. Two SIs which have to be compared are thus fed with the same input $\vec{p}$ and then compared with regard to their outputs and gains. Therefore, we build the distance vector $\vec{d}$ of the two given vectors

$$\overrightarrow{q_1{}^\circ\lambda_1} = (q_{1_1}, \dots, q_{1_n}, \lambda_{1_1}, \dots, \lambda_{1_k})$$

and

$$\overrightarrow{q_2{}^\circ\lambda_2} = (q_{2_1}, \dots, q_{2_n}, \lambda_{2_1}, \dots, \lambda_{2_k}).$$

However, the calculation of the distance of the single elements has to be adopted slightly, since we are operating on cyclic groups. Therefore, the distance between 0 and 1 should be the same as between 7 and 0 within $\mathbb{Z}_8$. In order to regard this characteristic, we define a special type of subtraction, leading to the minimal distance between two single elements. Figure 11 shows an example in $\mathbb{Z}_8$. Here, the distance between 2 and 7 should be 3 instead of 5 (which would have been the result of a classic subtraction).

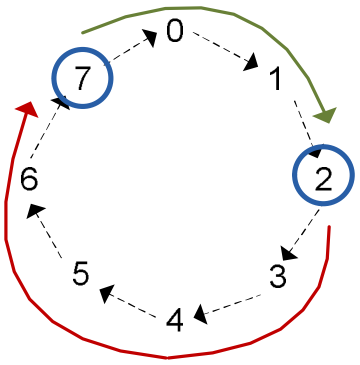


Figure 11: Minimal distance between elements in cyclic groups

We thus define the subtraction $-$ as $\min\{|a-b|, n-|a-b|\}$ for $a, b \in \mathbb{Z}_n$. By this definition, the maximal distance between two elements is $n/2$ for a domain with size n.

In order to abstract from the respective domain sizes, we normalize every difference by dividing it by half of the domain size. (As mentioned above, the biggest possible difference between two values within a domain is half of the domain space since we operate on cyclic groups. The normalized difference between two values will thus be between 0 and 1, where 0 denotes the equivalence of both values and 1 the biggest possible distance). In the following, we assume that all q_i and λ_i are already normalized. We then derive the distance between two SIs by building the norm of the distance vector:

$$\left|\vec{d}(q_1{}^\circ\lambda_1 - q_2{}^\circ\lambda_2)\right| = \sqrt{\left((q_{1_1} - q_{2_1})^2, \dots, (q_{1_n} - q_{2_n})^2, (\lambda_{1_1} - \lambda_{2_1})^2, \dots, (\lambda_{1_k} - \lambda_{2_k})^2\right)}.$$

Thus, the result of one run is a distance value $\left|\vec{d}(q_1{}^\circ\lambda_1 - q_2{}^\circ\lambda_2)\right|$.. Depending on the parameter *simRuns*, multiple input vectors are randomly generated and their output vectors are compared. The *distance index* is given by the mean value of all norms of the distance vectors.

This procedure also enables the comparison of SIs generating a different number of outputs. Consider the two exemplary parts of SIs depicted in Figure 12. Here, the first SI is capable to generate all required outputs while the second fails to create the output connected to port 2.

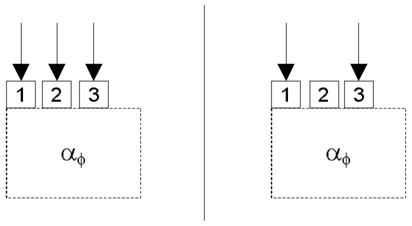


Figure 12: SIs generating varying outputs

We penalize this deviation by assuming the biggest possible distance to the created output, i.e. by adding half of the domain size. For instance, if the first SI has created an output of 12 within the domain $\mathbb{Z}_{100}$, we include 62 as output of the second SI. The normalized distance is thus evaluated as 1 normalized. (I.e. $\min\{|62-12|, 100-|62-12|\}$ = 50 divided by half of the domain size, i.e. 50 for $\mathbb{Z}_{100}$, evaluates to 1.)

2.2.5.3 *Example*

Let two service individuals *A* and *B* have a matching set of inputs $\overrightarrow{p_{A,B}} = (x{:}\,\mathbb{Z}_{10}, y{:}\,\mathbb{Z}_{52})$. We start the first simulation run with the random values $x = 5$ and $y = 22$. Assume *A* generates the output $\overrightarrow{q_A} = (12{:}\,\mathbb{Z}_{100})$ and the gains $\overrightarrow{\lambda_A} = (\lambda_1, \lambda_2)$ while *B* computes $\overrightarrow{q_B} = (33{:}\,\mathbb{Z}_{100})$ and the gain $\overrightarrow{\lambda_B} = (\lambda_2)$. We thus have to build the distance between $\overline{q_A \circ \lambda_A} = (0.24, 1, 1)$ and $\overline{q_B \circ \lambda_B} = (0.66, 0, 1)$. (Note that both vectors are already nomalized with regard to their domain size.) We derive the distance

$$\left|\overline{d(q_A \circ \lambda_A - q_B \circ \lambda_B)}\right| = \sqrt{-0.42^2 + 1^2 + 0^2} = 1.18,$$

which is also abstractly depicted in Figure 13, assigning the output to the x-axis and the gains to the y- and z-axis, respectively.

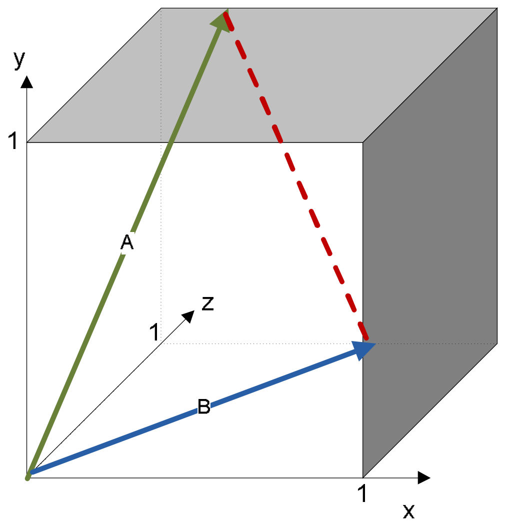


Figure 13: Distance between two SIs

Within deliverable D3.4.1, we will present extensive simulation results based on the setting introduced in this section.

3. Refinements of self-management concepts

3.1 Semantic injection in BIONETS Service Architecture

In the BIONETS project there is no intention to develop a new semantic approach, it is not a BIONETS objective and many other projects and organization are working on it (see appendix B). It has been considered useful to consider the research trends in distributed ontologies and semantic reasoning, and see how this can be applied to the BIONETS service architecture, i.e. Service Cell and Individual.

The semantic introduction means use of an ontology to be applied to data and services. Modern distributed systems frequently use autonomous software agents that use ontologies to communicate. If each agent owns an ontology or knowledge base and these ontologies change over time, every communication attempt requires merge and revision processes, i.e. the problem of providing seamless, integrated access to such sources has become a major research challenge: ontology mapping, alignment and merging. An ontology is an engineering artifact: it is constituted by a specific vocabulary used to describe a certain reality, plus a set of explicit assumptions regarding the intended meaning of the vocabulary.

Thus, an ontology describes a formal specification of a certain domain, a shared understanding of a domain of interest and formal and machine-manipulable model of a domain of interest.
The introduction of an ontology, of an ontology language and consequently of semantic reasoning functionalities can be obviously helpful in BIONETS as it would support several features:

- Openness – everybody can act as a provider or consumer of services. Openness is an essential necessity to ensure the success of the BIONETS platform.
- Heterogeneity – services are created in isolation from one another, thus interoperability is an issue. Content can appear, change, or disappear in an uncontrolled fashion.
- Distribution – there is no central control of services. Services can appear, change or disappear at any time in an uncontrolled fashion and without a central authority.
- Semantics can enable a new generation of the architecture, in which information has machine-processable and machine-understandable semantics.
- Discovery: the task where the client is interested in getting a specific service. Given a goal and some distributed service repository determine the set of relevant services along multiple dimensions:
  - Capability (service semantics/functional aspect)
  - Non-functional properties (Quality of Service Parameters, Provider)
  - Choreography (how to interact with the service)

  In a world of myriad of services it may cost too much to find the best solution. Pragmatic approaches in service discovery will focus on utility, i.e., stop the search process when a service is found that is “good enough” to fulfill a request. Also, it is unrealistic to assume that semantic descriptions of services are correct and complete, i.e. duplicate the functionality of a service at the description level.
- Mediation: the heterogeneity as inherent characteristic of BIONETS:
  - heterogeneous terminology
  - heterogeneous languages / formalisms
  - heterogeneous functionalities
  - heterogeneous communication protocols and business processes

3.1.1 Ontology Reasoning

Given the key role of ontologies in many applications, it is essential to provide tools and services to help users:

- Design and maintain high-quality ontologies, i.e..:
  - Meaningful — all named classes can have instances
  - Correct — captured intuitions of domain experts
  - Minimally redundant — no unintended synonyms
  - Richly axiomatised — (sufficiently) detailed descriptions
- Answer queries over ontology classes and instances, e.g.:
  - Find more general/specific classes
  - Retrieve individuals/tuples matching a given query
- Integrate and align multiple ontologies

On the ontologies it is necessary to build an ontology reasoning. In BIONETS context the choice made has been to adopt OWL or RDF (a simpler version). They are a W3C standard and DL (Description Logic)-based ontology language: OWL constructors/axioms are restricted so the reasoning is decidable. RDF(S) provides basic relational language and simple ontological primitives. It is necessary to have confidence in a reasoner and OWL DL benefits from many years of DL research providing:

- Well defined semantics
- Formal properties well understood (complexity, decidability)
- Known reasoning algorithms
- Implemented systems (highly optimised)

In Annex C, all the details on reasoning capabilities applying RDF or OWL can be found. Ontology provides more capabilities than datatypes, that are in any case important in the Semantic Web ontologies and applications, because it is needed to represent, in some way, various “real world” properties such as size, weight and duration, and some other complex user defined datatypes. Reasoning and querying over datatype properties are important and necessary if these properties are to be understood by machines.

For instance, e-shops may need to classify items according to their sizes, and to reason that an item which has height less than 5 cm and the sum of length and width less than 10 cm belongs to a class, called “small-items”, for which no shipping costs are charged. Accordingly the billing system will charge no shipping fees for all the instances of the “small-items”class.

Various Web ontology languages, such as RDF(S), OIL, DAML+OIL and OWL, have witnessed the importance of datatypes in the Semantic Web. All of them support datatypes. Description Logics (DLs: http://dl.kr.org/), a family of logical formalisms for the representation of and reasoning about conceptual knowledge, are of crucial importance to the development of the Semantic Web. Their role is to provide formal underpinnings and automated reasoning services for Semantic Web ontology languages such as OIL, DAML+OIL and OWL. Using datatypes within Semantic Web ontology languages presents new requirements for DL reasoning services. Firstly, such reasoning services should be compatible with the XML Schema type system, and may need to support many different datatypes. Furthermore, they should be easy to extend when new datatypes are required.

Considering a “River” class, require that the value of the “length-kmtr” property be 1.6 times that of the “length-mile” property. An extension of a language such as OWL to support the use of n-ary datatype predicates can be imagined:

```
<owl:Class rdf:about="River">
<rdfs:subClassOf>
<owl:NaryRestriction>
<owl:onProperties rdf:parseType="Collection">
<owl:DatatypeProperty rdf:ID="length-kmtr">
<owl:DatatypeProperty rdf:ID="length-mile">
</owl:onProperties>
<owl:allTuplesSatisfy rdf:resource="unit:kmtrsPerMile"/>
```

```
</owl:NaryRestriction>
</rdfs:subClassOf>
</owl:Class>
```

3.1.2 Lightweight Semantics and distributed reasoning for the BIONETS Service Architecture

Different approaches are currently under study in distributed environments like BIONETS's. In contrast to the so-called global approach, in which reasoning with multiple semantically related ontologies is performed in a global knowledge base that encodes both ontologies and semantic mappings, in the BIONETS approach a distributed reasoning approach is proposed in which reasoning is the result of combination via semantic mappings of local reasoning and "chunk" of distributed ontologies (and in a first approach also tagging or folksonomies are allowed) [5].

An important requirement for dynamic collaboration and semantic interoperability in open distributed systems is to have light semantic description of each component, and each component should implement a local semantic matchmaker which is responsible for the evaluation of the semantic affinity. Due to the dynamicity and variability of collaboration no centralized authority can assume or acquire a comprehensive view. To enable information processing and content retrieval in a distributed context with a multitude of autonomous ontologies, appropriate matching techniques are required to determine semantic mappings between concepts of different ontologies that are semantically related [6].

Each U-Node has its own local ontology (metadata) describing its contents to be shared with others. A dynamic enrichment of the local knowledge with new network knowledge (ontology mappings) acquired from outside is possible due to the interactions with other components. A selection service is required based on the query/answer paradigm to dynamically choose the relevant components based on the semantic affinity of their contents (semantic matchmaking).

The following foundations characterize the formation of distributed semantic annotation and sharing for autonomic components:

- **Ontology-based component description**. Each component exposes to the system an ontology (distributed and light - even incomplete -) which provides a semantically, more or less rich, representation of the services that the component exposes to the network, in terms of concepts, properties, and semantic relations.
- **Query-based interactions on a Semantic Data Space**. Each component interacts with the other components by submitting semantic selection queries in order to identify the potential components that can satisfy the request and by replying to incoming queries whether they can satisfy a given request.
- **Semantic matchmaking and reasoning capabilities**. Each component implements its own semantic knowledge as an ontology, and a distributed semantic matchmaker/reasoner for matching ontologies in order to find which concepts match in different ontologies and at which level. This role is played by specific Mediators.

On the basis of different approaches to semantically annotate the components, i.e. the semantic richness of services and data, of the system, there are also different semantic reasoning approaches. A surface matching is defined when considering only the names of concepts. Surface matching is suited for dealing with high-level, poorly structured ontological descriptions (i.e. a linguistic approach like a vocabulary or WORDNET). An intermediate matching is defined to consider both concept names and concept properties. With this model, a more accurate level of matching and reasoning is possible by taking into account not only the concept names but also information about the presence of properties and about their cardinality constraints. A more sophisticated matching model is possible when considering concept names and the whole context of concepts, and also semantic relations (as applying OWL).

3.1.2.1 Ontology evolution and adapatation

In order to develop such an approach, BIONETS should take in consideration ontology management, i.e. the whole set of methods and techniques that are necessary to efficiently use multiple variants of ontologies from possibly different sources for different tasks. Therefore, an ontology management system should be a framework for creating, modifying, versioning, querying, and storing ontologies.

Ontology modification is accommodated when an ontology management system allows changes to the ontology that is in use, without considering the consistency.

Ontology evolution is accommodated when an ontology management system facilitates the modification of an ontology by preserving its consistency.

Ontology versioning is accommodated when an ontology management system allows handling of ontology changes by creating and managing different versions of it.

In the BIONETS Service Framework ontology evolution concerns the capability of managing the modification of an ontology in a consistent way. An ontology may change because the domain or the user needs have changed or simply because the shared conceptualization has been modified. Ontology evolution can be defined as the timely adaptation of the ontology to changing requirements and the consistent propagation of changes to the dependent concepts. The goal of ontology evolution is to augment the background knowledge in the existing domain ontology to better classify extracted object descriptions. The ontology is enriched by adding concepts, properties, and/or semantic relations or by modifying existing ones, in order to produce a new ontology version capable of providing interpretation of the new multimedia resource.

A modification in one part of the ontology may generate subtle inconsistencies in other parts of the same ontology, in the ontology-based instances as well as in depending ontologies and applications. This variety of causes and consequences of the ontology changes makes ontology evolution a very complex operation that should be considered as both an organizational and a technical process. It requires a careful analysis of the types of the ontology changes that can trigger evolution as well as the environment in which the whole ontology evolution process is realized. [7], [8], [9]

Although evolution over time is an essential requirement for successful application of ontologies, methods and tools to support this complex task completely are missing (an example but not complete is http://kaon.semanticweb.org/). This level of ontology management is necessary not only for the initial development and maintenance of ontologies, but is essential during deployment, when scalability, availability, reliability and performance are absolutely critical.

The BIONETS Deliverable D3.2.2 [15] presented operations in order to work on lightweight and distributed semantics, and are sketched hereafter, but the BIONETS framework (specifically the Mediator) should foresee also some mechanism for ontology evolution consistency (i.e. reasoning, merging and matching). The operations are related to:

- Merge concepts.
- Replace several concepts with one and aggregate all instances.
- Extract subconcepts.
- Split a concept into several subconcepts and distribute properties among them.
- Extract superconcept
- Create a common superconcept for a set of unrelated concepts and transfer common properties to it.
- Extract related concept.
- Extract related information into a new concept and relate it to the original concept.

While *Insert*(t_1,t_2) provides an operation for extending local knowledge, *Remove*(t) can delete knowledge that is no longer necessary or valid. In case multiple semantic tags are identified to describe the same functionality, the *Unify*($t_1,\ldots,t_k$) operation allows to merge these different notions; in connected computing environments, this functionality can be provided by other and more general ontologies. The most important operation enabling the incorporation of remote knowledge is provided by the *Join(t, n_1, n_2)* operation. In case two DSTs (Distributed Semantic Tree) of nodes n_1 and n_2 both possess a tag *t*, the subtree from n_2 beginning at *t* can be appended to the DST of n_1. If this operation entails the presence of a service *s* at two different vertices on one path towards the root, the corresponding lower level link is kept while the more general one is removed. Therefore, nodes can extend their knowledge either temporally or permanently by external knowledge. Service link operations enable the insertion and deletion of semantic descriptions for services (*Insert(s, t)*, *Remove(s, t)*). Moreover, a service can be assigned a more concrete or more general description by the *Move(s, t_1, t_2)* operator.

3.1.3 *Semantic* Support for BIONETS evolutionary service lifecycle

3.1.3.1 Semantically-based Automatic Service Composition

A Service Cell should be characterized by a distributed semantic annotation. The Semantic Data Space should be able to deal with a semantic query, i.e. a semantic query of search of information (services or data). The services and data needed by a Service Cell will be searched through the Semantic Data Space. The Mediator will be able to handle with semantic reasoning and evolution (as described in the previous paragraph).
If a Service Cell needs some data and services to handle its job those are found through a Semantic Data Space search and selection. In this way an automatic service composition though semantics can be realized chaining those services and data. The Mediator (providing semantic reasoning), in order to find data and services can provide: match of semantic concepts, align data and service concepts, merge data and service concepts, reason on semantic concepts for data and service (inheritance, relationships).

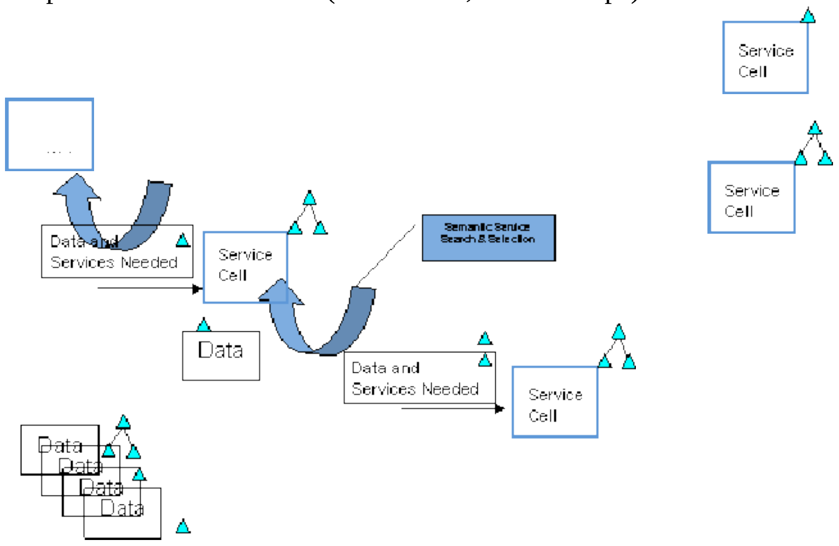


Figure 14: Semantic Composition.

3.1.3.2 Automatic Selection: gain driven

According to the semantic service description, it is possible to apply the selection and binding of a service on the basis of the concept of goal (or gain: see BIONETS Deliverable D3.2.3 [4]) w.r.t. its semantic description (see next paragraph). A semantic annotations of gains can be used and then make the reasoning based on semantic (similarity, sub-classes, …). In case a match is not found on the Semantic Data Space, a Mediator can handle reasoning in order to find another compatible solution which needs adaptation (change of names, similarities, etc.).

3.1.3.3 Ontology evolution/adaptation: an example

In the BIONETS Service Architecture there can be several Service Cells providing different services, and services with semantic similarities.

If the application needs a Service providing a Temperature, a semantic request on the Semantic Data Space is placed through its interface, e.g. a search request can be entered for a place where the "Temperature is hot". Then the Mediator that is in charge of dealing with semantics should be able to map and align all the concepts about 'hot': i.e. 32C, 90F, etc.

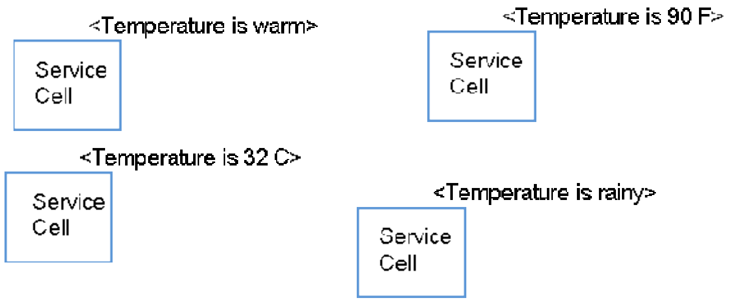


Figure 15: Distributed Semantic Annotation.

3.2 Semantic Service Specification

This section contains a proposed extended and adaptation of the OWL-S ontology that is suitable for a BIONETS Service Description.

The idea is to start from OWL-S [11], which guarantees a good and standardized base and a lot of background and specification activities are currently on going. Mainly, the idea is to develop and study a possible incremental approach in service description and semantic annotation of services suitable for BIONETS. All the descriptions refer to RDF or OWL, and the semantic reasoning approaches (specific tools are currently available on the web) can be applied to all the components that are specified in the following sections.

Not all the aspects can be covered and are needed in BIONETS: a service description covering some of the aspects that seem more relevant for BIONETS will be sketched, specifically taking in consideration an incremental approach.

The paragraph containing the possible adoption/adaptation will outline a pragmatic way for specifying BIONETS Service Descriptions.

3.2.1 OWL-S Ontology

The Semantic Web [11] should enable access not only to content but also to services on the Web. Users and software agents should be able to discover, invoke, compose, and monitor Web resources offering particular services and having particular properties, and should be able to do so with a high degree of automation if desired. Powerful tools should be enabled by service descriptions, across the Web service lifecycle. OWL-S (formerly DAML-S) is an ontology of services that makes these functionalities possible. The structure of the ontology is of three main parts:

- ServiceProfile - *What does the service provide for prospective clients?*

  It represents a semantic description (abstract) of the service. It presents also part of the information in the ServiceModel.

- ServiceModel - *How is it used?*

  It describes at an abstract level how the service works, with I/O (parameters), preconditions of the service execution, effects, and the modules the service is composed of. *Process* is the main class that models the service execution. The section 3.2.2.3 on adoption of the Process class for BIONETS presents a possible way of using it.

- ServiceGrounding - How does one interact with it?

  A grounding provides the needed details about transport protocols. Instances of the class Service have a support property referring to a Service Grounding. The Service Grounding will not be covered in this specification phase, the actual BIONETS implementation can will refer to it. The OWL-S approach can be adopted for BIONETS, as the different viewpoints on the Service Description are useful, and help in categorizing a service. Then how this can be used in BIONETS and which aspects are relevant will have to be analyzed

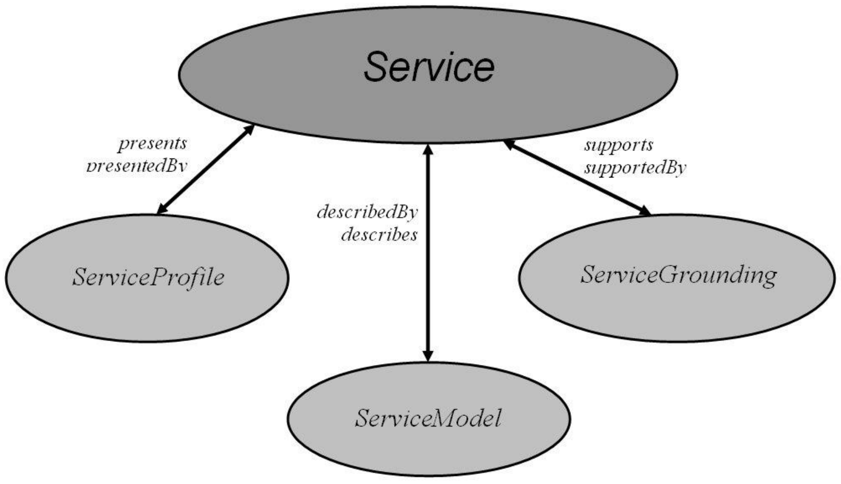


Figure 16: Service Representation in OWL-S.

3.2.2 An OWL-S ontology adaptation for BIONETS

3.2.2.1 Requirements

One important question is how well ontology languages represent the different aspects of BIONETS services. During discovery, it is essential to describe the capabilities of services, where such capabilities specify the overall functionalities of the service, and what the service does or provides. Expressing capabilities is essential for service providers so they can express what they achieve with their services. At the same time, the ability to express service capabilities is also essential for service requesters that can express what functionalities are expected from the providers. Ultimately, expressing capabilities allows the whole interaction process to be goal-directed, so that the client can achieve the results it needs.
The representation of capabilities goes beyond the specification of the functionalities, and it should include the specification of the conditions under which the service works as expected, and what conditions it imposes on the requester. Computational limitations or bandwidth limitations, and even range limitations, may be very constraining in mobile computing and are an integral part of the specification of the capabilities of the service. Similarly, the media used by the service, be it video streaming or text messaging, contribute to the specification of the capabilities of the service.
In addition to representing the different aspects of the capabilities of the Service, a number of additional features, often called service parameters, need to be represented. These service parameters include things such as the cost model of the service, whether the service uses a pay-per-use model or a flat fee or whether the access to this service is restricted to a specific class of users.

Since services have many different aspects, there is the need for a flexible representation schema that allows the service providers to express all the information that they need to express about their services, and the service requesters to express exactly what kind of services they seek. To support such flexibility, any service representation schema should be based on an inference mechanism that allows the extraction of knowledge that is implicitly stated in the description, but not explicitly exposed, and the derivation should allow the service requesters to “reason” about the service specifications that they receive.

Each service is more than just a process, it provides a well-defined functionality, it uses a specific communication interface, a payment modality, and is characterized by additional characteristics that are beyond the simple definition of a process. Specifically, for most of BIONETS services some properties need to be specified:

- Comm_Channel: the communication channel that is used to transmit the content between the service and the mobile platform. Using this property it is possible to specify whether the service uses Bluetooth or UMTS or any other protocol, and the restrictions both in terms of range and bandwidth.
- Cost_Model: the cost model used to charge for the service: flat rate, a fee per use, or free.
- Media: the type of media that is used to transmit the information to the platform. For example, the media may be simple text, sound or video.
- Device_Requirements: the service may make some assumptions about the device that is used to display the information: screen size and resolution, memory, processing speed, etc.
- Additional subclasses can be specified adding properties for specific types of services. For example, entertainment services need to specify the rating on the type of content that they display, and the expected audience. Location services need to specify the method of location, since the different methods have different degrees of precision.

For BIONETS service description the idea is to use the Service Profile (mainly for non-functional features) and Atomic Process description (for the functional description). Each field of the service description can refer to semantic or only data and data types. The Atomic Process refers to the functional part of the service; in BIONETS it can be stated to have all these features in the service description.

For the grounding features there will be a reference to the actual service specification. The semantics of a service composition in the first step will be the sum of the service descriptions of which the service is composed of. It is not possible to have an automatic service composition at this stage of the project and research results without human support.

3.2.2.2 Semantic Profile Specification for BIONETS

The specification activity presented in the document is based on the ontology Profile.owl, a subclass of ServiceProfile, defined in Service.owl [11] . The starting point of this work was taking into account the main characteristics encompassed by the profile ontology of OWL-S and their suitability to the BIONETS domain. The results are a number of extensions (listed in the following paragraphs) crafted to obtain a better semantic description of a BIONETS service in order to capture BIONETS service requirements [13].

Annex A reports a full description of the Profile Ontology in OWL-S, or better a proposed BIONETS extension for the Profile Ontology. It can be helpful in order to see all the aspects that should be covered.

The following example provides some extracts of a full OWL-S specification. It takes in consideration the service profile of a send SMS service.

There can be different approaches in BIONETS. In the first phase this profile can be expressed also as simple XML file: OWL and ontologies can be avoided. The different fields can simply refer to keywords (e.g. WORDNET), or type systems instead of ontologies. RDF or OWL can be added incrementally for specific purpose. Obviously the representation will be less flexible in the reasoning mechanism.

On the semantic information different kinds of operations are possible: matching, or reasoning on the semantic information (if they are part of the same class, restrictions on properties, link between properties, and mapping between concepts).

<!-- send SMS service. -->
```
<service:Service rdf:ID="SendSms">
    <service:supports>
      <grounding:XXXGrounding rdf:ID="SendSms_XXXGrounding">
    </service:supports>
<hasProcess rdf:resource="&process-c;#sendSms"/>
<service:presents>
<profile:Profile rdf:ID="SendSms_Profile">
       <service:presentedBy rdf:resource="#SendSms"/>
       <profile:serviceName
       rdf:datatype="&xsd;#string">SendSms</profile:serviceName>
```

```
        <OWL:externalProvider
        rdf:datatype="&xsd;#boolean">false</OWL:externalProvider>
        <profile:contactInformation rdf:resource="-"/>
        <profile:textDescription rdf:datatype="&xsd;#string">Tilab send SMS
        platform.</profile:textDescription>
        <OWL:isFree rdf:datatype="&xsd;#boolean">true</OWL:isFree>
        <profile:serviceParameter rdf:resource="&profile-c;#ClientSide"/>
        <profile:serviceParameter rdf:resource="&profile-
        c;#MessagingDevice"/>
        <profile:serviceParameter rdf:resource="&profile-c;#Consumer"/>
        <profile:serviceParameter rdf:resource="&profile-c;#Model-1-4"/>
        <OWL:serviceQuality rdf:resource="&sQual-c;#available"/>
        <OWL:serviceQuality rdf:resource="&sQual-c;#guaranteted"/>
        <OWL:serviceQuality rdf:resource="&sQual-c;#std-reliability"/>
        <OWL:serviceQuality rdf:resource="&sQual-c;#std-speed"/>
</profile:Profile>
</service:presents>
    <OWL:hasProvider rdf:resource="&provider-c;#Tilab"/>
  </service:Service>
```

3.2.2.3 Process

Then the ‘Process’ aspect of a service is reported. The starting point is the Process.owl defined by the OWL community. The specification using an ontology is not a trivial task. For this reason in BIONETS it would be useful to use a ‘light’ approach and incremental approach in order to cover if not all the aspects at least some of them.

```
  <process:AtomicProcess rdf:ID="sendSms">
          <process:name rdf:datatype="&xsd;#string">sendSms</process:name>
           <!-- sendSms's input parameters. -->
           <OWL:hasInput rdf:resource="&parameter-c;#smsBody"/>
           <OWL:hasInput rdf:resource="&parameter-c;#phoneNumber"/>
           <OWL:hasInput rdf:resource="&parameter-c;#destNumber"/>
           <!-- sendSms's output parameters. -->
           <!-- sendSms's effects. -->
           <process:hasEffect rdf:resource="&effect-c;#SendSms"/>
  </process:AtomicProcess>
```

In the Process definition, it can be worthwhile in the BIONETS approach to start for the effect (or gain) approach. HasInput, hasOutput can refer to type systems/interfaces, while for effects that are used on the SDS for searching services, semantics can be applied.

Here some individuals of the class Effect on SendSms, SendInstantMessage follow.

```
  <!-- Send's individuals. -->
  <effect:Send rdf:ID="SendInstantMessage">
       <effect:textDescription rdf:datatype="&xsd;#string">Send an Instant
       Message (IM).</effect:textDescription>
       <rdfs:label>SendInstantMessage</rdfs:label>
       <rdfs:comment rdf:datatype="&xsd;#string">As an example consider
       operations "sendMessage" and "sendMessageSimple" offered by the
       StarSIP-based InstantMsg WS.</rdfs:comment>
  </effect:Send>
  <effect:Send rdf:ID="SendSms">
       <effect:textDescription rdf:datatype="&xsd;#string">Send a
       SMS.</effect:textDescription>
       <rdfs:label>SendSms</rdfs:label>
       <rdfs:comment rdf:datatype="&xsd;#string">Useful to model mobile
       services.</rdfs:comment>
  </effect:Send>
  <effect:Send rdf:ID="SendMms">
       <effect:textDescription rdf:datatype="&xsd;#string">Send a
       MMS.</effect:textDescription>
       <rdfs:label>SendMms</rdfs:label>
       <rdfs:comment rdf:datatype="&xsd;#string">Useful to model mobile
       services.</rdfs:comment>
  </effect:Send>
  <effect:Send rdf:ID="SendEmail">
```

```
      <effect:textDescription rdf:datatype="&xsd;#string">Send an e-
      mail.</effect:textDescription>
      <rdfs:label>SendEmail</rdfs:label>
      <rdfs:comment rdf:datatype="&xsd;#string">Useful to model Internet
      services.</rdfs:comment>
  </effect:Send>
```

3.3 Migration Support for Evolutionary Services

In D3.2.2 [15] we presented a high-level specification of the replication and migration support in the lifecycle of evolutionary services, i.e. services that can evolve. In this chapter we refine the notion of service migration within the context of atomic service cells.

We first clarify the definition of service migration. Then we present a high-level scenario of service mobility in the BIONETS environment and analyze the requirements for different components and functionalities. We also present research done in the specific topics related to the service migration like decision-making mechanisms.

3.3.1 Introduction & Motivation

The BIONETS Service lifecycle needs to support service migration. The coexistence of multiple service instances in the same node has a limited usefulness (e.g. to create a certain level of redundancy for fault tolerance purposes). The services need to, e.g. in case of resource shortage in a host device, be able to automatically migrate themselves to a new host device which is offering better “living conditions”. Various migration policies can be implemented. For example, services may migrate randomly, they may migrate toward their users, or they may migrate to achieve a load-balance among hosting computers [16], [17].

The BIONETS environment hosts two kinds of service entities; Service Cells and Service Individuals, both are mimicking the behavior of biological entities. As Chapter 3.3.2 in D3.2.2 [15] presented, for Service Individuals the migration procedure is more complex compared to Service Cells. While the migration of Service Individuals is still part of our ongoing work, we started with specifying the migration of Service Cells.

The research challenges of Service Cell migration contains many different separate topics, e.g. high-level logic like automatic spread and migration of services, defining the geographic distribution of populations of services according to available resources, demand, cost and other constrains. Also other more general topics, not necessarily related to service migration, need to be tackled, like service autonomy leading to solutions for service migration decision-making, service migration execution, context-based service adaptation to environmental changes, and service discovery.

3.3.2 Service Mobility Demonstration scenario

The purpose of the following demonstration is to present a high-level overview of a scenario in the BIONETS environment where a service migrates (and replicates also) from the hosting node to another node.

3.3.2.1 Demonstration overview

Figure 17 presents a step-by-step illustration of the service mobility scenario, where a Service Cell X_{01} resides in a U-node B and then migrates to another U-node A. The figure is composed of three phases: a) Initial situation, b) service replication & migration and c) service adaptation.

In phase a) the Service Cell X_{01} is located at the U-node B and it is offering its services to U-node A (or actually to the user of U-node A). The events leading to phase a), like the formation of the node island or the service discovery process, are not in the scope of this demonstration scenario.

In phase b) U-node A (or the user carrying U-node A) is moving away from the U-node B. The Migration Mediator (MMediator) of Node-A is constantly monitoring (between some intervals, e.g. 60s) and evaluating the services utilized in that node for assuring the best possible level of service to the user. In this case, the MMediator of Node-A monitors the fading connectivity with U-node B and going through a decision process it decides that it should request a replica of the Service Cell X_{01} from U-node B and migrate it to U-node A, as there is increased risk of lost connectivity. The MMediator of U-node B creates a replica X_{02} and negotiates with MMediator in U-node A about the migration of the Service Cell X_{02} to U-node A.

Phase c) presents the situation after Service Cell X_{02} has moved and adapted to U-node A. The MMediator of U-Node A takes care of the seamless crossover between old service and new one so there are no breaks in the service provisioning to the user.

Figure 18 presents the sequence diagram for the service replication and migration scenario presented above. The diagram illustrates the sequence of the simplified message transfers for the negotiations between the mediators handling the service replication and migration functionalities.

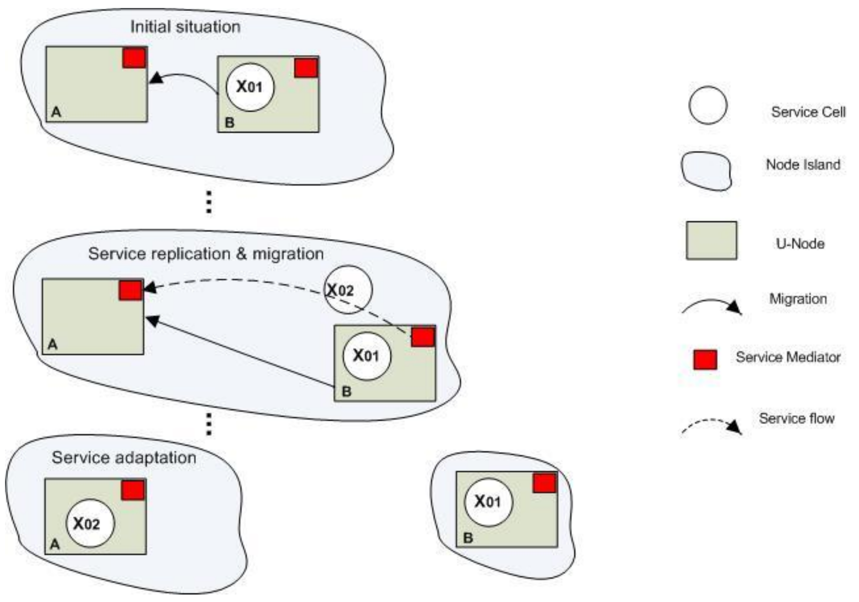


Figure 17: Step-by-step illustration of the service mobility scenario

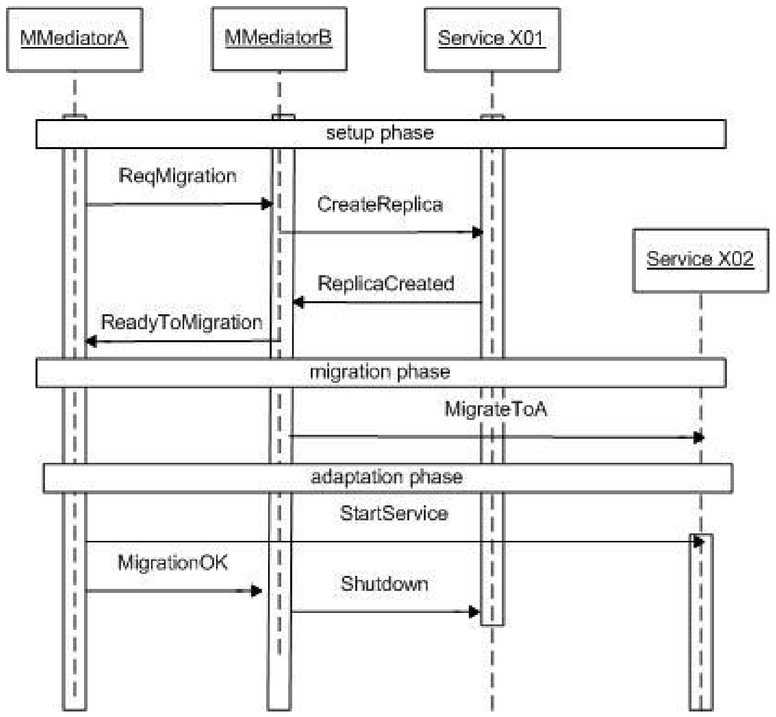


Figure 18: Sequence diagram of the service mobility scenario

3.2.2.2 Analysis of different components and functionalities

Associated with service migration there are many kinds of challenges related to the replication and actual migration of one Service Cell. We need to solve some fundamental challenges; e.g. what does the service or mediator need to know about the underlying network, network resources, and terminal resources in order to produce successful migration from one node to another? In addition, we need to research what (automated) functionalities we need when the migration is about to happen and what entity provides them? Or what is the impact on the BIONETS service architecture? For example, what kind of requirements does the Service migration procedure cause for the service discovery, deployment, and evolution (addressing and naming policies) etc.?

In the following the different phases, components and functionalities of the BIONETS platform are needed in each phase are discussed further.

Prerequisite

In order to have a functional service migration, we need to address some general service framework issues, like service identification, discovery and communication framework. The services need to be identified uniquely. Besides the functional description the service should offer also non-functional information. E.g. in the migration case it would be vital to know the service's device requirements. The MMediator needs to "see", not just the services in their node, but also all the services in their node island (inside connection range). The services and the mediators also need to be able to communicate and interact with each other. As presented in D3.2.2 [15], the Interaction Framework component enables the services and mediators to interact in a peer-to-peer manner with the other members of the system by submitting discovery queries (semantically-based) in order to identify the potential members of a given "aggregation" and by replying to incoming queries whether it can join a given aggregation. The routing in the network based on context-aware packet forwarding and the end-to-end communication can be based on e.g. distributed hash tables [18], [19].

Setup & Migration process:

In the BIONETS service framework mediators have a central role. As was discussed in D3.2.2 [15] in the first version of the BIONETS service framework the autonomic functions are implemented in Service Mediators and not in the actual Services. Mediators were introduced to implement service life-cycle functionalities. Mediators abstract autonomic functionalities from services (in order to allow them to remain lightweight and able to run on resource-constrained devices). U-nodes can host a variety of different kinds of mediators, each providing their own set of autonomic functions to the services. In the service migration scenario, presented above, we have a migration mediator component, which is specialized in offering the autonomic service migration functionalities to Service Cells. The MMediator is responsible for ensuring the best possible level of service flow produced and/or consumed in the host node. For example, in the migration scenario presented above, MMediator is monitoring the connection link between nodes A and B when a weakening signal level triggers a decision process in the MMediator to react to the changing situation. The MMediator performs the fitness evaluation based on the network information, node statistics (context and routing/forwarding information) and controls and initiates the migration process. The actual decision and control logic can be implemented using a utility function based on the measurements monitored from the CPU load, battery consumption, network connectivity and service fitness (user evaluation, number of requests, etc.). The decision process is presented in more detail in Section 3.2.3. After the decision process is ready (in the demo scenario the decision was to migrate the service) the actual implementation phase is started. The MMediator A makes a request to the MMediator B for the service migration. The MMediator B doesn't move the original service itself, but creates a replica of the service and migrates it to node A. The service platform of the U-node needs to provide the functionalities to inject new service into the node and control mechanism for activating them.

Adaptation

After the service is migrated to the new host node, the mediator of the originating node hands over the control of the service to the new host node's mediator. The mediators communicate together for ensuring the seamless service flow during the hand-over. The service platform (mediators) of the host node needs to adapt the newly injected service to the new surroundings by for example advertising it to the other services and nodes, but also allocating its resources, like CPU-time.

3.3.3 Decision-Making Mechanisms for Service Migration

The key element of the MMediator is a decision-making module that invokes the migration process. This module has the following main tasks:

- Identification of the options, for example identification of the possible destination service platforms
- Identification and evaluation of the relevant parameters for each option, for example CPU load level and battery level
- Weighting of the parameters. The weighting can be based on reliability information or simply subjective judgments
- Utility function evaluation. The overall utility value for each option is a weighted average of the utility values calculated for each parameter
- Making a rational decision, that is, selecting an option with the maximal utility value.

There exist two critical questions in the tasks of MMediator: how to compute a utility value for each parameter? And how to determine weights for each parameter? If a parameter is purely technical, i.e. evaluation of its value does not involve a subjective component at all, it is usually easy to find a suitable utility function representation. For example, in its simplest form battery level can be seen as this kind of parameter, the fitness value can be represented by a linear utility function. An example of more complex parameters is QoS information which should evaluate how the user feels about the quality of service. Determination of the utility value for such a parameter requires psychological background knowledge of user behaviour and extensive user experiments. Usually utility functions take smooth forms, for example logistic sigmoids and the free parameters of these function classes are determined by user studies.

In the first version of the BIONETS service framework the utility values are weighted with simple subjective judgments and these weights are fixed. A more elaborated way to implement the weighting is to utilize reliability information. For example, if the CPU usage information from a platform is old and we are not able to update this information, the associated weight for the parameter should also be small.

4. Conclusion and Next Steps

In this document we have reported the current state of our research in several topics researched in the BIONETS WP3.2. We refined the service evolution aspects, containing the specification of service evolution on composition level as well as a framework for the cooperative evolution of services. This deliverable also presented the refinements of the self-management concept, containing the specification of semantic injection in the BIONETS service architecture, semantic service specification, and a summary of the achievements in migration support for evolutionary services.

For the service evolution research topic further work could consist in combining the approach suggested in 2.1 and 2.2 in order to let hierarchical and complex services evolve at each level, including evolution of their dependencies. This research perspective is very challenging and probably some restrictions will also have to be applied for the approach to be feasible in practice.

In the next phase for the semantic service specification and semantic injection in the BIONETS service architecture, the semantic, ontology specification and reasoning mechanisms are to be applied in the specific use case that is being developed in WP3.3.

In the next phase of the work, for migration support of BIONETS evolutionary services, the presented demonstration scenario will be verified with simulations and also implemented in real life devices. Also we are looking into integrating the service migration demonstration with the context aware routing/forwarding scheme from WP1.2.

5. References

[1] Heiko Pfeffer, David Linner, Stephan Steglich, and Ilja Radusch (ed.) Specification of Service Life-Cycle, BIONETS Project Deliverable D3.2.1, January 2007.

[2] Daniele Miorandi (ed) Architecture, Scenarios and Requirements Refinements, BIONETS Project Deliverable D1.1.2, July 2007.

[3] F. Baude, L. Henrio, and P. Naoumenko, "A Component Platform for Experimenting with Autonomic Composition". First International Conference on Autonomic Computing and Communication Systems (Autonomics 2007). Invited Paper. ACM Digital Library.

[4] Françoise Baude and Ludovic Henrio (ed.) Graph-based Service Individual specification: Creation and Representation, BIONETS Project Deliverable D3.2.3, December 2007.

[5] S. Castano, A. Ferrara, S. Montanelli, "Ontologies and Matching Techniques for Peer-based Knowledge Sharing". Conference On Advanced Information Systems Engineering (CAISE 2003)", Klagenfurt/Velden, Austria, June 2003

[6] L. Serafini and A. Tamilin, "DRAGO: Distributed reasoning architecture for the semantic web". Technical Report T04-12- 05, ITC-irst, December 2004.

[7] M. Klein, N. F. Noy, "A component-based framework for ontology evolution". Technical Report IR-504, Department of Computer Science, Vrije Universiteit Amsterdam, March 2003.

[8] L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. "User-driven ontology evolution management". In 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02), Siguenza, Spain, Oct. 1–4, 2002.

[9] S. Castano, A. Ferrara, and S. Montanelli, "Evolving open and independent ontologies," Journal of Metadata, Semantics and Ontologies, 2006.

[10] P. Haase and Y. Sure, "State-of-the-art on ontology evolution," Institute AIFB, University of Karlsruhe. EU-IST Integrated Project (IP) IST-2003-506826 SEKT, Deliverable D3.1.1.b (WP3.1), August 2004.

[11] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. Mcllraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara, "Owl-s: Semantic markup for web services". http://www.daml.org/services/owl-s/1.1/overview/, 2004.

[12] T. Berners-Lee, J. Hendler and O. Lassila, "The Semantic Web". Scientific American, May 2001.

[13] M. Paolucci, W. Goix, A. Andreetto, M. Luther, M. Wagner, "Representing Services for Mobile Computing using OWL and OWL-S: An Initial Investigation", in Proc. WWW Service Composition with Semantic Web Services,2005.

[14] http://www.w3.org/TR/rdf-schema/ - RDF Vocabulary Language 1.0.

[15] Lidia Yamamoto (ed.) Specification of Service Evolution, BIONETS Project Deliverable D3.2.2, July 2007.

[16] J. Suzuki and T. Suda, "A middleware platform for a biologically inspired network architecture supporting autonomous and adaptive applications". IEEE Journal on Selected Areas in Communications, 23(2): p. 249-260, 2005.

[17] D. Milojicic, F. Douglis, Y. Panedeine, R. Wheeler, and S. Zhou, "Process migration". ACM Computing Surveys, 32(3), 2000.

[18] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment". IEEE JSAC, 22(1):41-53, Jan. 2004.

[19] Y. Kadobayashi, "Achieving Heterogeneity and Fairness in Kademlia". In Proceedings of the International Symposium on Applications and the Internet Workshops (SAINT 2004), p.546-551, Jan. 2004.

[20] S. Alouf, I. Carreras, D. Miorandi and G. Neglia, "Embedding Evolution in Epidemic-Style Forwarding", in Proc. of BioNetworks (IEEE MASS) 2007.

[21] S. Alouf, I. Carreras, A. Fialho, D. Miorandi and G. Neglia, “Autonomic Information Diffusion in Intermittently Connected Networks”, to appear in Autonomic Computing and Networking, Springer, 2008.

[22] J.A. Foster, “Evolutionary computation”. Nature Genetics Reviews, vol 2, pp. 428-436 June, 2001.

6. ANNEX A: Profile Ontology

A profile has to be associated to a service in a bidirectional way, than Profile inherits from ServiceProfile the property:

service:presentedBy : service:ServiceProfile -> service:Service - O (any number)

it is associated at the *inverse property* in the class Service

service:presents : service:Service -> service:ServiceProfile - O (any number)

Profile presents the following three properties:

serviceName : Profile -> String - D (exactly one)

textDescription : Profile -> String - D (exactly one)

contactInformation : Profile -> owl:Thing – O (any number)
[usually a URI list]

The notation D indicates that it is a *Data Property*, i.e. it is a property in the range of a basic class like integer, String, etc., while O stands for *Object Property*, i.e. it is a property in the range of a class instance of an ontology.

Then 5 properties related to the functional description of the service; generally it would be a good thing that the profile would be consistent with the ProcessModel of the service, i.e. that the following parameters are consistent and refer to the IOPE (Input Output Preconditions Effects) created in the class Process.

hasParameter : Profile -> process:Parameter - O (at most one)

hasInput : Profile -> process:Input - O (any number)

hasOutput : Profile ->process:ConditionalOutput – O (any number)

hasPrecondition : Profile -> process:Precondition - O (any number)

hasEffect : Profile -> process:ConditionalEffect – O (any number)

has_process : Profile -> process:Process - O (at most one

The classes Input and ConditionalOutput are subclasses of Parameter, that is never instantiated. They can be used to model conditions. It has to be underlined that ConditionalOutput has a subclass UnConditionalOutput, and ConditionalEffect has a subclass UnConditionalEffect, those concepts will be used in the first specification of WS.

The Profile class offers optional parameters like:

serviceParameter : Profile -> ServiceParameter - O (any number)

serviceCategory : Profile -> ServiceCategory - O (any number)

The ServiceParameter class describes a set of characteristics of a service; e.g. for the cost of a service. ServiceParameter represents the domain of the properties:

serviceParameterName : ServiceParameter -> String - D (exactly one)

sParameter : ServiceParameter -> owl:Thing - O (exactly one)

The range of sParameter is too wide and it has to be restricted in relation to the real services that are to be modeled[3].

The ServiceCategory class allows mapping services on specific categories in a taxonomy that can be used as a reference; ServiceCategory is the domain of the following category.

[3] `owl:Thing` is the basic class of an OWL ontology; a property of range `owl:Thing` can have as a value any instance of any class.

categoryName	: ServiceCategory -> String	-	D (exactly one)
taxonomy:	ServiceCategory -> String	-	D (exactly one)
value	: ServiceCategory -> String	-	D (exactly one)
code	: ServiceCategory -> String	-	D (exactly one)

Following the DAML Coalition guidelines, the property taxonomy and code should refer to an official taxonomy, like the UNSPSC (United Nation Standard Products and Services Code System), or the NAICS (North America Industry Classification System), and to a specific item in the taxonomy. The categories provided by those taxonomies are at such a high level that they need to be more detailed in order to cover BIONETS service or network or software component.

In order to describe Services BIONETS it has been necessary to modify OWL-S and to extend in a special purpose way the classes Service.owl, Profile.owl, Process.owl and Grounding.owl. A new OWL document has been created that import those files and introduces new concepts and properties.

The extensions presented are grouped in the following sections on generic information of the service, another one on the .specialization introduced for serviceParameter, a section related to a bidimensional categorization of the service, and one on the properties of the quality of service offered. Then the extension of the class Service and the concept of provider of services.

6.1 Generic information

The information in the profile have been extended in order to consider also the following properties:

isFree	: Profile -> Boolean	-	O (exactly one)
externalProvider	: Profile -> Boolean	-	O (exactly one)

These information are in the class Profile and not part of ServiceParameter as they will be part of all profiles and object of frequent query.

6.2 Service Parameter

In order to restrict the range of the property sParameter, in the namespace profile context the class MyServiceParameter has been created. It is the root of an ontology of service parameters, the individuals will be used for the values of sParameter. In future the MyServiceParameter could be taken out of the class Profile in order to be specialized. There will not be instances of individuals of type MyServiceParameter, but only descendants.

- ContentType domain of two data property, that should have one of the following value.

  content : ContentType -> {streaming, conversational, messaging}

  media : ContentType -> {audio, textual, video}

  This entity is used to model the way by which the service is used, or the type of multimedia contents. content and media will be presented together as specific individual of ContentType; e.g. video-call will be described as the individual video-call : content = conversational; media = {audio U video}.

- Cost is the domain of the data property costModel, of the value:.

  costModel : Cost -> {model-1, model-2, model-3, model-4}

  The different model-i represents a generic cost model.

- ServiceActors domain of the data property

  Actors : ServiceActors -> {Person2Person, Person2Content,
  Content2Person}

  ServiceActors instances allow modeling the interaction way between the different participants to a service.

- ServiceInvocation domain of the data property activator.

activator : ServiceInvocation -> {clientBased, serverBased}

The *initiator* of the service, server or client (push vs. pull).

- Session.

  type : Session -> {call, session, subscribe} ⊂ String

  mode : Session -> {continuous, periodic} ⊂ String

Session represents the interaction type between user and service.

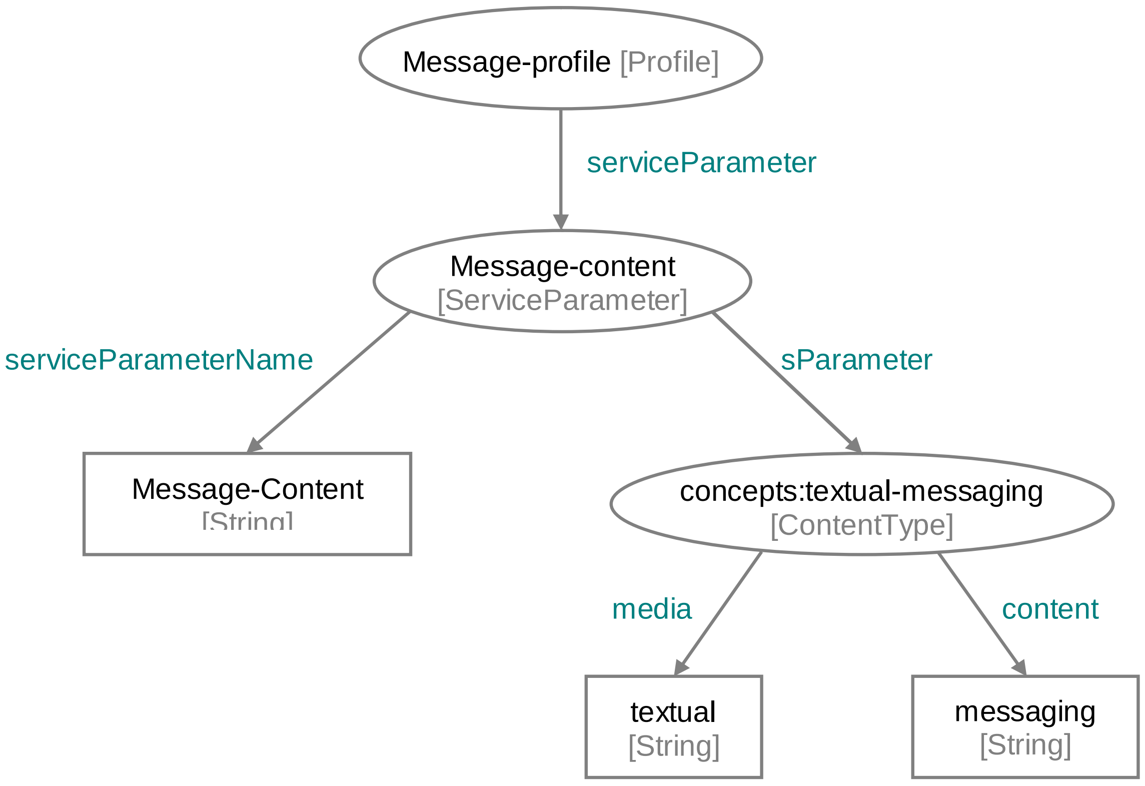


Figure 19: Service Profile for BIONETS.

The Figure19 represents the use of the MyServiceParameter ontology, the *RDF graph* is used to represent a service to envoy a text message. The ellipse is an instance of "complex" concept; the rectangle represents instances of primitive type; the arrow has the name of the property of the RDF triple.

6.3 Categorization of the service

In order to categorize a service three properties have been added to the class Profile.

serviceFunction	: Profile ->	ServiceFunction	–	O (at least one)
mainServiceContext	: Profile ->	ServiceContext	–	O (exactly one)
relatedServiceContext	: Profile ->	ServiceContext	-	O (any number)

The classes ServiceFunction and ServiceContext (that derive from MyServiceCategories) represent a service from two viewpoints:

- a viewpoint related to the technological aspect of the service, like *messaging services* as the Instant Messagging *or localization services*, etc.

- another viewpoint is related to the target of the service (i.e. its application context). Some services are related to the *communication*, other to *entertainment* (games, wine&food, etc), etc. It is previewd a mainServiceContext (the main target of the service) and relatedServiceContext.

MyServiceCategories, ServiceContext and ServiceFunction do not define a property and they are not instantiated but are related to the concepts of the classes that derive from them and below explained.

ServiceFunction presents two subclasses currently:

- Location the domain of the data property locationServices, should assume one on the value in the list.

  LocationServices : location -> {mobileLocationServices_MLS,

  webResourceLocation, … }

- Messaging as explained before.

  messagingServices : Messaging -> {istantMessaging, … }

ServiceContext has the following subclasses:

- CommunicationServices the domain of the data property communication should assume one on the value in the list.

  communication : CommunicationServices -> {messages,

  voiceCall, videoCall, … }

- EntertainmentServices, as before.

  entertainment : EntertainmentServices -> {news, games,

  wine&food, hotel, … }

The ServiceFunction and ServiceContext enable the classification of the services in respect to the technology and target viewpoint, and they need to be further detailed.

6.4 Quality of service

The OWL-S ontology should take in consideration some information on the quality of the service that describes, for this reason the class Profile has been specialized through the property:

serviceQuality : Profile -> ServiceQuality - O (any number)

the range is a certain number of instances of the class ServiceQuality which is the domain of the following property that should have these specific values.

qualityRate : ServiceQuality -> {low, standard, high}

The property is inherited from all the subclasses of ServiceQuality (currently ServiceGuarantee, ServiceReability, ServiceSpeed).

An example: the InstantMessagging service guarantees the delivery of messagges to the recipient, the individuals Message_WS-profile serviceQuality will have the value safe : qualityRate = high instance of ServiceGuarantee and reable : qualityRate = high instance of ServiceReability.

The Figure 20 presents the schema of the ontology Profile after all the extensions, it is not an RDF graph, but just a representation schema.

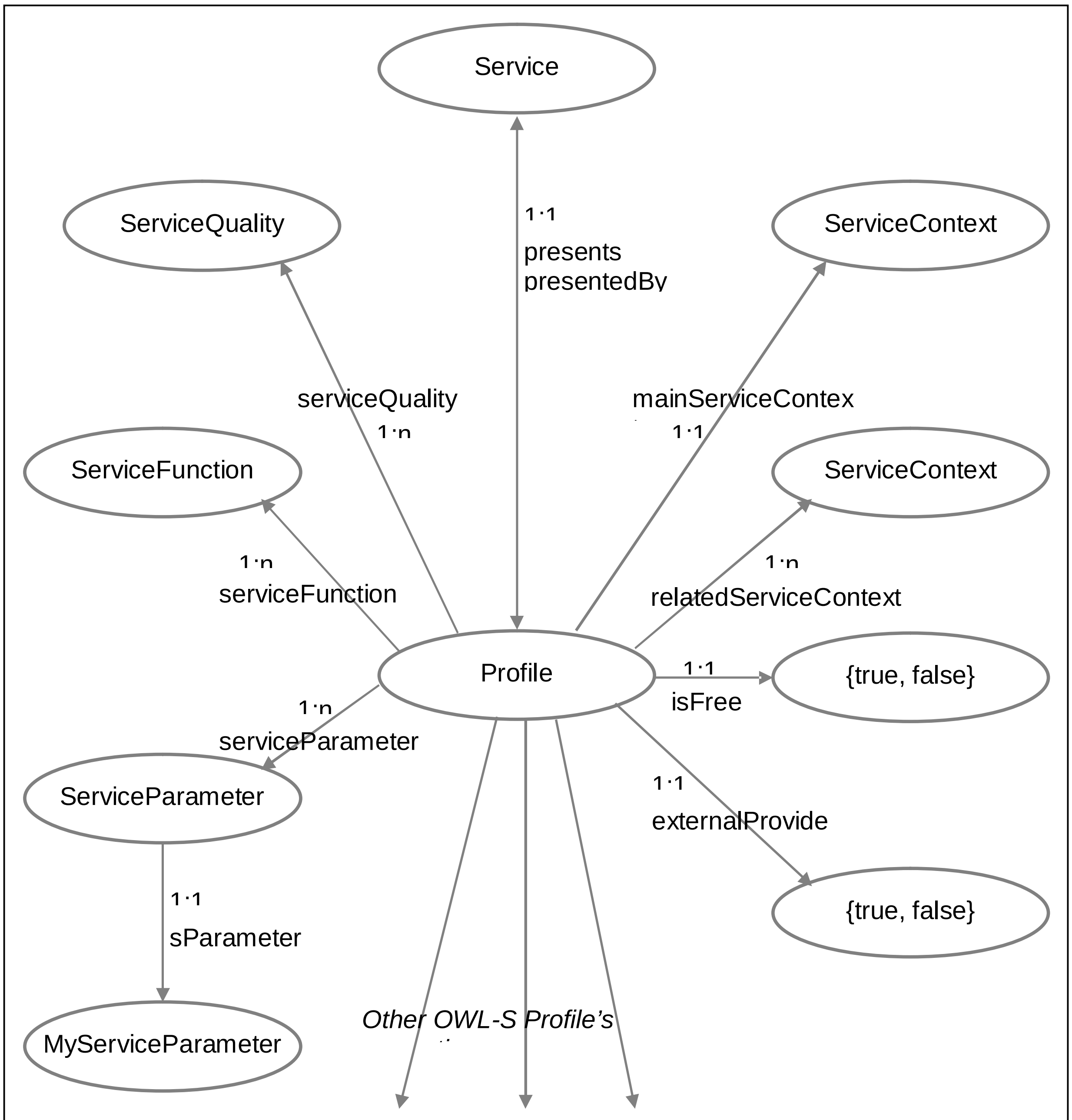


Figure 20

6.5 Provider Class

The ontology service:ServiceProvider (part of Service, and not of Profile) has been created, in order to capture the provider of a service, from wich provider:Provider class derives and it is the domain of the following properties:

providerDescription	: Provider -> String	-	D (exactly one)
providerName	: Provider -> String	-	D (exactly one)
providerContacts	: Provider -> owl:Thing	–	O

[usually a URI list]

With the introduction of the Provider class the property service:providedBy and service:provides can assume the following values:

service:providedBy	: service:Service	-> Provider	- O (exactly one)
service:provides	: Provider	-> service:Service	- O (any number)

7. Annex B: State-of-the-art and current research trends

Semantic and use of ontologies in BIONETS are considered a requirement for the service description. Ontologies in different formalism like RDF [13], folksonomies, tagging are raising in the web and for this reason it has been considered an aspect to be taken in consideration in BIONETS. The BIONETS context of a disconnected network and different devices interacting with different services on top of them should address the specification of services and also their semantic in order to enable the other requirements like self-configuration and dynamic selection and binding.
In this section it will be given a brief overview on other initiatives and projects and how the semantic is applied on those contexts. This can provide the flavour on how BIONETS can take inspiration and apply semantics.
REWERSE Reasoning on the Web with Rules and Semantics, a FP6 IST European project (terminated at begininning 2008).
The objective of REWERSE was to establish Europe as a leader in reasoning languages for the Web by networking and structuring a scientific community that needs it, and by providing tangible technological bases that do not exist today for an industrial software development of advanced Web systems and applications, to develop a coherent and complete, yet minimal, collection of inter-operable reasoning languages for advanced Web systems and applications.
BOEMIE (Bootstrapping Ontology Evolution with Multimedia Information Extraction) is an IST 6th Framework Programme Project which has started in March,1 2006. BOEMIE will pave the way towards automation of the knowledge acquisition process from multimedia content which nowadays grows with increasing rates in both public and proprietary webs, and will break new ground by introducing and implementing the concept of evolving multimedia ontologies. The project is unique in that it links multimedia extraction with ontology evolution, creating a synergy of enormous yet unrealized potential.
Mobile Ontology, the ontology work developed in the European IST-FP6 project SPICE. The Mobile Ontology is a higher-level comprehensive ontology for the mobile communications domain. The ontology is a machine readable schema intended for sharing knowledge and exchanging information both across people and across services/applications. In the scope of SPICE, the Mobile Ontology is designed to serve as data exchange format between the components of the platform, as well as reasoning tool.
Within the **On-To-Knowledge-Project** (http://ontology.ist-spice.org/index.html) have been developed tools and methods for supporting knowledge management relying on sharable and reusable knowledge ontologies.
In the Sensoria IST Project the purpose of the Sensoria ontology is to provide the participants in the Sensoria project with a common conceptual model for service-oriented computing. In particular, the aims are:

- to provide a terminological background and enhance the integration and the consistency of the work carried on in different work packages,
- to distil a basic set of concepts and aspects that have to be considered when designing a service oriented system. Such concepts and aspects define a guideline for the definition of ad hoc languages addressing the design of the various aspects of a service-oriented architecture.

The ontology consists of a glossary and a model. The glossary describes the main classes of the ontology by using the natural language. The model is based on UML (mainly on class diagrams).
The issue of semantic interoperability, integration, and coordination between heterogeneous content sources in distributed systems is addressed in several research fields, such as:

- Data integration: schema matching [Bernstein et al., VLDB 2001, Lenzerini et al., ACM PODS 2002, Castano et al., IEEE TKDE 2001]

- Semantic Web: ontology matching [Bouquet et. al., CONTEXT 2003, Halevy et al., WWW 2002, Castano et al., JODS 2006]
- Schema-based P2P networks: digital resource discovery [Broekstra et al., WWW 2003, Nejdl et al., WWW 2002, Castano&Montanelli, KER 2007]
- Grids and Grid-DBs: knowledge sharing [Cannataro et al., WWW 2003, Decker et al., WWW 2003]

This kind of rationale is useful for the BIONETS approach, i.e. distribution and evolving context.
Ontology Alignment Evaluation Initiative (http://oaei.ontologymatching.org/). The increasing number of methods available for schema matching/ontology integration suggests the need to establish a consensus for evaluation of these methods. The Ontology Alignment Evaluation Initiative is a coordinated international initiative to forge this consensus.
The goals of the Ontology Alignment Evaluation Initiative are:

- assessing strength and weakness of alignment/matching systems;
- comparing performance of techniques;
- increase communication among algorithm developers;
- improve evaluation techniques;
- most of all, helping improving the work on ontology alignment/matching.
- through the controlled experimental evaluation of the techniques performances.

Between others it can still be mentioned **Tones** (Thinking ONtologiES) is an Information Societies Technology 3-year STREP FET project financed within the European Union 6th Framework Programme; and **SEKT** (researching SEmantic Knowledge Technologies) is an EU-IST Integrated Project (IP) started in january/2004 and with duration of 36 months.
KAON [http://kaon.semanticweb.org/] is a tool suite for ontology management. One of the functionalities of the KAON is the support of ontology evolution. The management of versions, however, is basically manual. It is the user who has to specify the effects of the changes done over the ontology.

8. Annex C

The section reports the reasoning mechanisms that can be achieved using RDF or OWL [11].

8.1 RDF/RDFS Reasoning Capabilities

Type inheritance through rdfs:subclassOf. For example, the facts

- (rdf:type Morris Cat)
- (rdfs:subClassOf Cat Mammal)

imply the fact

- (rdf:type Morris Mammal)

Reflexivity of rdfs:subPropertyOf and rdfs:subclassOf. For any rdf:Property p, the fact (rdfs:subPropertyOf p p) is inferred.
For any Class C, the fact (rdfs:subClassOf C C)is inferred.

Type inference through rdfs:range and rdfs:domain constraints. For example, the facts

- (rdfs:domain teaches Teacher)
- (rdfs:range teaches Student)
- (teaches Bob Scooter)

imply the facts
(rdf:type Bob Teacher)
(rdf:type Scooter Student)

Transitivity of rdfs:subClassOf and rdfs:subPropertyOf. For example, the facts

- (rdfs:subClassOf Dog Mammal)
- (rdfs:subClassOf Mammal Animal)

imply the fact (rdfs:subClassOf Dog Animal)
Similarly, the facts

- (rdfs:subPropertyOf parent ancestor)
- (rdfs:subPropertyOf ancestor relative)

imply the fact (rdfs:subPropertyOf parent relative)

8.2 OWL Reasoning Capabilities

OWL reasoning capabilities include the RDF/RDFS reasoning capabilities as well as the following capabilities. In all references to reasoning with rdfs:subClassOf, the same type of reasoning is done with owl:subClassOf.

- Enforcing transitivity of owl:TransitiveProperty. For example, the facts
- (rdf:type ancestor owl:TransitiveProperty)
- (ancestor Sue Mary)
- (ancestor Mary Anne)

  imply the fact
  (ancestor Sue Anne)
- Semantics of owl:SymmetricProperty is enforced.
- Reasoning with owl:inverseOf. For example, the facts
- (owl:inverseOf parentOf hasParent)
- (parentOf Goldie Kate)

  imply the fact
  (hasParent Kate Goldie)
- Inheritance of disjointness constraints. For example, the facts
- (owl:disjointWith Plant Animal)

- (rdfs:subClassOf Mammal Animal)
  imply the fact
  (owl:disjointWith Plant Mammal)
- When an owl:sameAs relationship is asserted or inferred between two entities that are known to be classes, an owl:equivalentClass relationship is inferred between the classes. Similarly, when an owl:sameAs relationship is asserted or inferred between two entities that are known to be properties, an owl:equivalentProperty relationship is inferred between the classes. For example, the facts
- (owl:sameAs Human Person)
- (rdf:type Human rdfs:Class)
- (rdf:type Person rdfs:Class)
  imply the fact
  (owl:equivalentClass Human Person)
- All the subclasses of a given class are disjoint with the class's complement. For example, the facts
- (owl:complementOf Animal NonAnimals)
- (rdfs:subClassOf Mammal Animal)
  imply the fact
  (owl:disjointWith Mammal NonAnimals)
- A complicated bit of reasoning about owl:complementOf captured by following KIF axiom.
- (=> (and (owl:complementOf ?c1 ?c2)
-         (rdfs:subClassOf ?c3 ?c1)
-         (rdfs:subClassOf ?c4 ?c2)
-         (owl:complementOf ?c4 ?c5))
-     (rdfs:subClassOf ?c3 c5))
- Inferring owl:sameAs relationships via owl:FunctionalProperty and owl:InverseFunctionalProperty. For example, the facts
- (rdf:type mother owl:FunctionalProperty)
- (mother Joe Margaret)
- (mother Joe Maggie)
  imply the fact
  (owl:sameAs Margaret Maggie)
- If a class A is owl:oneOf a list of objects, say X, Y, and Z, then each of X, Y, and Z has rdf:type A.
- If an object is rdf:type an owl:hasValue owl:Restriction, then the object has the specified value for the specified property. For example, the facts
- (owl:onProperty RestrictionOrangeSkin skinColor)
- (owl:hasValue RestrictionOrangeSkin Orange)
- (rdf:type MrOompaLoompa RestrictionOrangeSkin)
  imply the fact
  (skinColor MrOompaLoompa Orange)
- If an owl:hasValue owl:Restriction restricts a particular property to a particular value, and an object has that value for that property, then the object has the Restriction as a type. For example, the facts
- (owl:onProperty RestrictionOrangeSkin skinColor)
- (owl:hasValue RestrictionOrangeSkin Orange)
- (skinColor MrOompaLoompa Orange)
  imply the fact
  (rdf:type MrOompaLoompa RestrictionOrangeskin)

- If an object is a rdf:type an owl:allValuesFrom owl:Restriction, and the object has values for the specified property, then the values are of the specified type. For example, the facts
- (owl:onProperty RestrictionCatChildren child)
- (owl:allValuesFrom RestrictionCatChildren Cat)
- (rdf:type Fluffy RestrictionCatChildren)
- (child Fluffy Cupcake)

  imply the fact
  (rdf:type Cupcake Cat)
- If an owl:someValuesFrom owl:Restriction restricts a particular property to a particular type, and if an object has some values of the specificied type for the specified property, then that object has the Restriction as a type. For example, the facts
- (owl:onProperty RestrictionIvyLeagueDegree degree)
- (owl:someValuesFrom RestrictionIvyLeagueDegree IvyLeagueSchool)
- (degree Mary Harvard)
- (rdf:type Harvard IvyLeagueSchool)

  imply the fact
  (rdf:type Mary RestrictionIvyLeagueDegree)
- If a property Q is owl:inverseOf of a property P, and P is an owl:TransitiveProperty, then Q is also an owl:TransitiveProperty.
- All of the elements of an owl:AllDifferent are owl:differentFrom each other.

OWL-DL has its foundations in Description Logics, which are decidable fragments of First Order Logic. For a particular task, a logic is decidable if it is possible to design an algorithm that will terminate in a finite number of steps (i.e., the algorithm is guaranteed not to run forever). For example, in Description Logic it is possible to write an algorithm that calculates whether or not one concept is a subclass of another concept, which is guaranteed to terminate after a finite number of steps. Because an OWL-DL ontology can be translated into a Description Logic representation, it is possible to perform automated reasoning over the ontology using a Description Logic reasoner. A Description Logic reasoner performs various inferencing services, such as computing the inferred superclasses of a class, determining whether or not a class is consistent (a class is inconsistent if it cannot possibly have any instances), deciding whether or not one class is subsumed by another, etc. Some of the popular Descrioption Logic reasoners that are available are RACER, FaCT, FaCT++, Pellet.