# Towards a comprehensive approach to spontaneous self-composition in pervasive ecosystems

Sara Montagna, Mirko Viroli, Danilo Pianini
DEIS–Università di Bologna
via Venezia 52, 47521 Cesena, Italy
Email: {sara.montagna, mirko.viroli, danilo,pianini}@unibo.it

Jose Luis Fernandez-Marquez
University of Geneva
Route de Drize 7, CH-1227 Carouge, Switzerland
Email: joseluis.fernandez@unige.ch

***Abstract*—Pervasive service ecosystems are emerging as a new paradigm for understanding and designing future pervasive computing systems featuring high degrees of scale, openness, adaptivity and toleration of long-term evolution. A key issue in this context is making certain patterns of behaviour emerge without any supervision or design-time intention, and a primary example is the fully-spontaneous composition of services, possibly at multiple levels. We argue that this can be successfully achieved only by a comprehensive approach exploiting together the main ingredients proposed so far in literature: *(i)* existence of intelligent components finding proper (semantic) matches of service descriptions, *(ii)* use of distributed evolutionary techniques to dynamically select appropriate ways of composing services, and *(iii)* approaches in which rating quality of composition is solely based on their successful exploitation. This proposal is presented through an example of spontaneous composition in crowd steering services.**

## I. Introduction

Services are one of the main actors in pervasive and ubiquitous computing. Composing services is a big issue in these contexts, providing new applications or high level services. Self-composition is even more challenging.

In literature some surveys and possible solutions have been proposed, but only few of them give concrete proposals for the design, implementation and evaluation of service composition. Moreover the composition of services is solved in a centralised way, solution which hardly deal with the large-scale distributed property of pervasive systems, or the composition is specified by the application developers, preventing the system to use new services that appear at runtime.

In this paper we frame/situate the problem of self-composition in the framework of pervasive ecosystems, defining what a service is and what composing services means. We then propose an approach to perform self-composition of services and demonstrate our ideas showing how gradient - a crucial service in pervasive system applications, in particular for those ones used for crowd steering – can be composed with other services. In particular our discussion focuses on the composition of gradient with services providing local levels of crowd, at the purpose of avoiding path crossing overcrowded areas. This example is modelled in terms of "Live Semantic Annotations" (LSAs) and eco-laws, that are the basic components of a pervasive ecosystem [20], and preliminary simulations, aimed at validating the approach, are run on top of Alchemist [14].

## II. Self-Composition approaches

A software service represents a functionality that can be provided on demand in order to create higher level software artifartefactsacts, such as applications or higher level services. The notion of service allows applications to be easily developed and to adapt by changing the different services, which the application is composed by, at run-time. Self-composition of services involves the automatic discovery of new services by composing available ones, the selection of services, the plan to execute them (i.e. services execution order), and the adaptation of the composed service when new requirements appear or a service disappears from the system.

*1) Service Composition in SOA:* The static character of traditional composition approaches such as orchestration and choreography has been recently challenged by so-called dynamic service composition approaches, involving semantic relations [5], and/or AI planning techniques to generate process automatically based on the specification of a problem [21]. Their basic goal is to analyse a description of the services to compose and compute a composition satisfying structural/behaviour/ontological compliance of the result of composition as can be deduced from information about the composites. One of the main challenges of these approaches is their limited scalability and the strong requirements they pose on the detail of service description.

*2) Evolutionary techniques:* Evolutionary approaches such as those based on Genetics Algorithms (GA) have been proposed as well for service composition, such as in [7], motivated by the need of determining the services participating in a composition that satisfies certain QoS constraints, which is known to be a NP-Hard problem. More generally, evolutionary approaches have been used in the field of autonomic computing to establish the set of norms, policies or rules that drive the system to the desired emergent behaviour [15]. We observe that this approach has a potential for service composition: in the same way as the genetic programming determine the proper execution of functions and their parameters, it could be used to find which services, their order of execution and the parameters that its service should receive.

*3) Competition-based approaches:* In order to tackle the potentially high number of different compositions that can arise in an open system, in [18] a mechanism of *coordination*

*reactions* for networked tuple spaces is proposed, showing how it can be applied to support competition and composition in a fully distributed setting—services opportunistically compose in those regions of the network where this results in a more competitive service. While all compositions can possibly take place, thanks to a positive/negative feedback only successful ones are meant to "survive" in the system, the others fading until becoming never actually selected. This is achieved by matching services with requests through a self-regulating dynamics inspired by prey-predator model of population dynamics [6], and by diffusing service tuples using a computational-field approach similar to the one presented in [12], [4].

## III. A COMPREHENSIVE APPROACH FOR PERVASIVE ECOSYSTEMS

As previous section emphasised, the issue of service composition is becoming wider and wider, and is now encompassing new research areas. Among them – pervasive computing, autonomic computing, robotics – we here focus on pervasive computing, and in particular, on the framework of pervasive ecosystems [22]. There, we understand and design a very large and dense pervasive computing system as a sort of substrate in which humans, institutions, software systems and devices, inject services of various kinds without an a-priori knowledge of the structure and behaviour of those already available and those that will be injected later. Such services diffuse in the network, and are situated and context-aware in the sense that they contextualise to (i.e., they will affect and be affected by) the situation in each *niche* of it. In turn, while in standard SOA a service is a centralised point of functionality, interacting with its clients by stateful protocols of message-passing actions, a service is here a more generalised concept: it is any activity, triggered by the intention of some localised software agent, that flows in the system updating the spatial and temporal configuration of other services, and generating a set of events ultimately perceived by the other agents in the system. Examples of such services include: the materialisation of data produced by all sensors available around, routing services able to interconnect devices based on their physic al, logical or social proximity, local/global advertisers of the availability of some content, situation recommenders capturing relevant information about a context, and so on. Typical application scenarios include pervasive display ecosystems, augmented social reality, and traffic control—the reader is deferred to [22] for a more deeper description of them.

### A. Abstract architecture

In particular we here focus on the SAPERE framework [19], an incarnation of the pervasive ecosystems paradigm based on the bio-chemical inspiration. This is based on the following abstractions: *(LSA)* the various software agents living in the ecosystem (whether they run on smartphones, sensors, actuators, displays, or any other computational device) have a uniform representation exposing any information about the agent (state, interface, goal, knowledge) that is pertinent

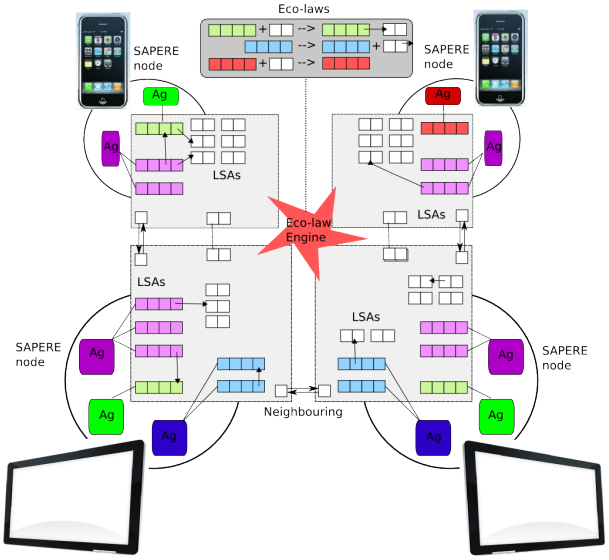


Fig. 1. An architectural view of a pervasive ecosystem.

for the ecosystem, which is called (LSA); *(LSA-space)* the LSAs of each agent are reified in a distributed space (called an "LSA-space") acting as the *fabric* of the ecosystem, and are stored in the portion of it that reside in a specific computational device; *(LSA bonding)* to make any agent act in a meaningful way with respect to the context in which it is situated, a mechanisms based on *bonds* (i.e., a reference from one LSA to another) is introduced: it is only via a bond that an agent can inspect the state/interface of another agent and act accordingly; *(eco-law)* while agents enact their *individual* behaviour by observing their context and updating their LSAs, *global* behaviour (i.e., global system coordination) is enacted by *self-organising* manipulation rules of the LSA-space, called *eco-laws*, executing deletion/update/movement/re-bonding actions applied to a small set of LSAs in the same locality in a chemical-resembling style.

Figure 1 shows an architectural view, based on the above abstractions, of a portion of an ecosystem featuring: two smartphones (carried by people) and two public displays forming a network of 4 computational nodes; a local LSA-space and some agents running in each node (e.g., recommendation agents, advertising agents, visualisation agents in displays, profile agents and sensor agents in smartphones); LSAs through which agents manifest (in colour); additional LSAs representing data, knowledge, and contextual information like the existence of neighbouring nodes (in white); bonds between LSAs; and a set of eco-laws executed by an underlying engine working over the global LSA-space.

In a more general case, one should think at a very large and mobile set of devices connected to each other based on proximity creating a distributed "space" – ideally a pervasive continuum – where LSAs form spatial structures evolved over time.

### B. A language of semantic chemistry

Inspired by the work in [20], we here present a language for expressing information as semantic annotations, and chemical-resembling rules as declarative manipulations of those annotations. We fully-ground it on frameworks and technologies for the Semantic Web, due to their support for openness and since they provide standard and well-accepted specification techniques. Namely, we inherit several syntactic and semantic aspects of RDF (Resource Description Framework [2]) and the SPARQL (query languages for RDF [16]) for coding rules: the main advantage of this choice is that off the shelf query engines (supporting execution of SPARQL queries and updates over RDF stores) and reasoners [17] can be used to support scheduling and execution of rules locally—hence directly leading to an implementation, though we shall not deepen this aspect in this paper.

In our framework, an annotation is a semantic tuple with a unique, system-wide identifier, and a content (description). This is realised as an RDF-like set of multi-valued properties, or equivalently, a set of triples "$s\ p\ o$" that consist of a subject (a tuple-id), a predicate (the property name, a Uniform Resource Identifier – URI) and an object (the assigned value, a literal or a URI). URIs are qualified by universally-accessible namespaces (using syntax `namespace:term`), typically containing ontologies defining certain aspects of a domain. By adopting notation N3 [1], an annotation is e.g.

```
id p v; q w1, w2, w3; r z1, z2 .
```

where `id` is the annotation-id, property `p` is assigned only to value `v`, property `q` is assigned to values `w1`, `w2`, and `w3`, and finally property `r` is assigned to values `z1` and `z2`—we will often omit the trailing dot in an annotation. The main advantage of this structuring of annotations with respect to syntactic tuples (sequences of values) is that it better scales with complexity of descriptions – as will be clearer below, e.g., we will not need to remember the meaning of $i^{th}$ value in a tuple, or mention variables for all tuple arguments when defining templates.

Following [8], rules are structured as chemical-resembling reactions, using a language of patterns specifically introduced to tackle semantic annotations (and using constructs inspired to SPARQL [16]). A rule is of the kind "`P+..+P --r--> Q+..+Q`". Elements `P` and `Q` are patterns of annotations, namely, annotations decorated with the following extensions:

- In place of each element of a triple one can use a variable `?V` (matching any value) or an annotated variable `?V(`*`filter`*`)` where *`filter`* is a predicate expression over `?V`: an annotated variable can match any value that makes the *`filter`* true. As an example, `?V(?V>5)` is such that `?V` can be substituted by any number greater than 5. Concerning filters, one can rely on any filter expression as defined in SPARQL. We sometimes also use as special filter for a subject `?T` an expression of the kind "`?T clones ?R`", meaning that annotation `?T` should have the same content of `?R` plus additional constraints specified by any following triples.
- In place of each element of a triple one can use an expression inside parenthesis, which the underlying engine should evaluate to a standard value. As an example, `(?V+5)` has a meaning when variable `?V` is bound to a value $n$, in which case the whole expression yields the result of adding 5 to $n$. Also for expressions we rely on SPARQL syntax.
- The object $o$ of a triple "$s\ p\ o$" can be prepended by a symbol "+", "-", or "=". The former (which is assumed by default when no symbol is prepended) means that the triple should exist, the second that it should not exist, and the latter that it should be the only one that exists for that subject and predicate ($s$ and $p$).
- When in a triple "$s\ p\ o$" we use a variable enclosed in square brackets `[?V]` in place of $o$, it means that `?V` will be bound to a list of objects $o'$. Then as usual, symbols "+", "-", or "=" can be prepended to mean that the corresponding triples "$s\ p\ o'$" exist, should not exist, should be exactly the list of triples for $s$ and $p$.
- For syntactic convenience, we also allow a pattern to consist solely of the source, meaning no further constraint on its triples is imposed.

The semantics of a rule is then simply understood as a pre/post-condition application. It consumes a set of *reactant annotations* matching left-hand side patterns and correspondingly produces a set of *product annotations* obtained by applying the post-conditions expressed in the right-hand side patterns. As in [8] transition rules also obey a numeric transformation rate `r`. Here it represents a Markovian rate in a continuous-time Markov chain (CTMC) system. Such a rate can be omitted, in which case it is assumed to be infinite, that is, the transition rule is executed with "as soon as possible" semantics. Execution of a transition rule amounts to atomically removing reactant annotations from the space and inserting product annotations back. The formal semantics of this rule language can be given by translation into SPARQL queries and updates, which we do not present here for the sake of space.

Our framework handles topological aspects as follows. First of all, each annotation carries a special property named `mid:#loc`, associated to the id of the location where the annotation is currently stored. Secondly, as in [8], the semantics of rules dictate that all reactant annotations should reside in the same location $l$, whereas product annotations can reside in $l$ or in any neighbour of it. Since it is stored in a property, the location of an annotation (and hence, annotation movement and diffusion) can be handled symbolically like any other property. Finally, we shall also assume that in each location there are some annotations (of type `mid:#neigh`, called *neighbour annotations*) specifically maintained by the underlying infrastructure to keep track of neighbours. For instance, the following annotation

```
:id314 mid:type mid:#neigh; mid:#loc :loc117;
```

```
mid:remote :loc118;  mid:distance "11.4";
mid:orientation "north-east"
```

expresses that in node `:loc117` there is the perception of neighbour node `:loc118`, in north-east direction and at estimated distance 11.4 (e.g., meters).

## IV. The self-composition issue in pervasive service ecosystems

Pervasive ecosystems naturally call for a radical shift in the way software is architected and developed. The concept of "software system" per se loses there its standard meaning of a monolithically designed/implemented/deployed artifact. It is rather a mash-up of services, and the development of a "new software system" simply translates into the development of additional services to be immersed in a scenario of existing ones, with the goal of composing with them and evolving their functionalities, thus making software live in a sort of "eternal beta" state [11].

The natural questions we have to answer in order to support this vision are hence: what can make services compose if they were not explicitly designed to do so? how can we decide "whether" and "how" two or more services should compose? how can such decisions be continuously reconsidered depending on the actual (spatial/temporal context) in which the composition is deployed? can we make multi-level composition seemingly work as well?

We refer to this problem as the self-composition problem for service ecosystems, where by "self-" we mean that the underlying middleware makes sure that services tend to compose in order to form meaningful new services, in a way that users of the ecosystems simply perceive only atomic and composite services that are actually useful in practice.

We argue that this vision cannot be fulfilled by the above-mentioned approaches taken in isolation, but instead require them to work altogether. Advanced (semantic) matching of services is key to decide whether two services deployed by third parties can be composed together. Evolutionary approaches seem the only solution to the problem of finely tuning the parameters used in the composition, preselecting, for instance, a subset of more promising compositions. And finally, competition of different compositions based on their actual exploitation appears instead as the only viable approach to measure the quality of a composition, hence promoting successful ones. On the other hand, understanding how they can work together in a comprehensive framework is very difficult, and generally depends on the specific platform and service model one adopts.

### A. Self-Composition with Gradient service

To ground our discussion of requirements and possible solutions, we here outline a paradigmatic example of composition in service ecosystems. We consider a crowd steering scenario, based on the idea of guiding people towards locations hosting events of interest in a complex and dynamic environment (using semantic matching with people's interests) without any supervision, namely, in a self-organised way. In particular,

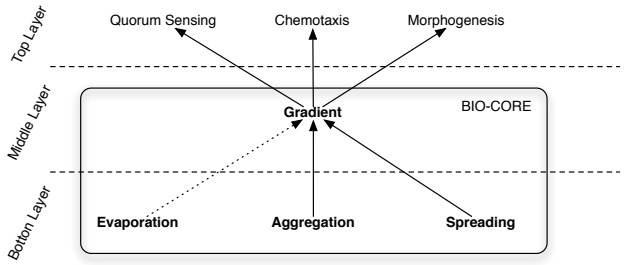


Fig. 2. Self-organising patterns and their relationships.

we consider a museum with a set of rooms connected by corridors, whose floor is covered with a network of computational devices (called sensor nodes). These devices exchange information with each other based on proximity, have sensors of various kinds, and hold information as LSAs (e.g., about exhibits currently active in the museum). Visitors exploring the museum are equipped with a smartphone device that holds their preferences. By interaction with sensor nodes, a visitor can be guided towards rooms with a target matching his/her interest, thanks to signs dynamically appearing on the smartphone or on public displays. This system is based on a gradient service, a key brick of self-organisation mechanisms (see Figure 2) that is typically designed to provide optimal paths to roam a distributed system even in the context of very articulated environments (a building with rooms and corridors, or a traffic scenario) and by dynamically adapting (namely, self-healing) to unpredicted situations such as sudden road interruption [8], [9]. Gradient starts from a source LSA located at an exhibition and spreads clones of it to all the nodes of the network by eco-laws of spreading and aggregation, basic building patterns as shown in Figure 2. In turn, as the gradient is stabilised, in each node the minimum distance value is kept, so that the shortest path to the source is automatically (and dynamically) defined [14]. In this way, a person willing to reach a point of interest simply "binds" to a gradient LSA, and public/private displays show signs to guide her to the source.

However, other services exist in the system, such as those provided by the many sensors available around. Among them we have the crowd detection service, enacted by the injection in each node on the floor of a crowd LSA reporting the crowd level sensed there (0 is no crowd, 1 is maximum crowd). Another service is the one that, given visitor preferences and profile, defines a set of other possible points (exhibitions, toilets, refresh points) the user can found interesting or useful. This service can be an external agent that is able to process visitor information, for instance defining related exhibitions, so as to provide, in the form of an LSA, a list of best points the user can pass by. Then we can have sensors for temperature, weather forecast, accelerometers etc. More generally, being a pervasive system an open system, new services may appear in the system, and autonomously be composed with existing ones, without a predefined schema. Only those compositions that create useful services will then survive. To exemplify

the idea, in the following we describe two self-composition problem we envisioned.

*1) Crowd composition:* This self-composition problem focuses on making the gradient service automatically compose to the crowd detection service so that the estimated distance to the source gets "penalised" in case of a high crowd value, which implies that the path computed towards a point of interest intrinsically takes into account situations of crowded areas dynamically emerging in the environment, properly circumventing them. In particular what we expect to the middleware to automatically do is: decide to compose the gradient with crowd sensing instead of with other (irrelevant sensors), and optimally tune how much crowd information should affect the computation of optimal path.

*2) Ad-hoc path composition:* If the gradient service is tagged with the point of interests crossed, it can be composed with the list of "best points" defined for each user, so that, in case of intersection, such gradient gets "reinforced" for instance diminishing its value, which then implies that the direction it comes from has more chance to be chosen, and paths which cross points interesting or useful for the user are likely to be followed. In particular what we expect to the middleware to automatically do is: decide to compose the gradient with the LSA representing the "best points", and optimally tune to which extent information on preferences should affect the computation of optimal path.

### *B. A prototype solution*

Starting from a number of services available in those nodes where sources of gradient services are located – all reified by an LSA that can be injected in different parts of the system but via chemotaxis reach gradient sources – the solution we propose is based on the following mechanisms:

**Composition.** Based on one or more "composition recommender" agents available in source nodes, all the available compositions are computed using techniques of advanced matching of interface/behaviour/ontology. These are ranked – according for instance to feedbacks from the users on the composite service, as discussed later – and the description of a limited number of them is reified in the space in the form of a set of LSAs. In the example of crowd steering proposed above, these compositions should result in a set of LSAs representing the source of different gradients, each one created composing the basic gradient with one or more services. We assume that the composition actuated, *i.e.* which services are composed and which parameters are used in the composition, is stored into a property of the gradient LSA, so that the same composition can be performed in all the locations of the system

**Contextualisation.** From the source, composite gradients and the basic one, are diffused. In each location they are contextualised with the local value of those services they are composed by. Contextualisations involving gradients normally result in a modification of the local value of distance according to a function $\delta$ specified in the service or defined by the composition agent. If services are not locally available, contextualisation does not happen and the distance value does not change. For instance, contextualise gradient with the crowd level perceived in each location at the purpose of discouraging paths crossing crowded areas. It means that distance should be increased and function $\delta$ looks like $\delta = k * c$ where $k$ is a function parameter (see later how to define its values), and $c$ is the local crowd level. The resulting effect is that the gradient service is available in the system in different forms (the basic one and the composite ones).

**Feedback.** Each gradient owns a property called "satisfaction", representing the feedback of users on the quality of the service itself. It can for instance evaluate the length of the path, the interest and so on: the higher the value is, the higher is the user satisfaction. Once people reach the source, they should then provide their feedbacks in the form of LSAs with predefined properties —such as length, comfort, beauty, interest. Marks for each property might be defined in a predefined range. From there one agent – or an application specific eco-law modelling the aggregation pattern – can then be in charge to compute the satisfaction value as a function of all the marks given – for instance, the average. The LSA of the correspondent gradient source is then modified by adding to the previous value of the satisfaction property the last computed. This updated value is then spread in the whole system autonomously as a result of gradient diffusion.

**Choice.** Users, once entering in the system, can probabilistically choose among the set of gradients available, grounding such decision mainly on the distance and satisfaction values. They will then follow the same composition for the whole path.

**Evaporation.** The satisfaction value can also be interpreted as the relevance value of the Evaporation pattern [8]. This parameter gets decreased until fading if not augmented by user positive feedbacks. Once it is equal to zero the composite service can be removed. This mechanism ensures that no satisfactory compositions are decayed.

**Evolution** If parameters are to be chosen for the composition – for instance parameter $k$ in $\delta = k * c$, representing how much the crowd level has to influence the distance value of the field –, they can be tuned by agents implementing evolutionary techniques. They are in charge to define the new population of parameters according to the fitness function that can be computed from the satisfaction value of the gradient itself.

The emergent idea is that, after a transition period, system makes available only those services that better fit user preferences. For instance, for those users interested in quickly reach the sources, only those gradients that ensure the lowest arrival time should survive.

## V. A FORMAL MODEL FOR GRADIENT SELF-COMPOSITIONS

To describe how self-composition of gradients can be built in the pervasive ecosystem framework outlined in Section III, we first introduce the implementation schema for gradients which we shall rely upon to formalise the self-composition task in the specific example of crowd service exploitation.

### A. The gradient service

Our strategy for modelling gradient features: *(i)* a continuous broadcast of information in each node as in [3], with latter information always overwriting previous one; *(ii)* propagation of estimated distance, computed by summing the contributions given by property `mid:distance` in neighbour annotation; *(iii)* propagation with no horizon (it is easy to impose a distance bound to the propagation of information); and *(iv)* after been sent to a neighbour, an annotation is subject to a contextualisation phase (in which e.g. aggregations are executed) before being spread again.

Such rules are reported in Figure 3 and are described in turn. Note that they refer to namespace `mid` for URIs related to middleware activities, and to `sos` for those concerning self-organisation aspects.

Rule [PUMP] is a transformation rule which takes a source annotation and creates a new annotation with distance set to $0$, used to start the spreading process. The source annotation should feature property `sos:type` assigned to value `sos:source` (we shall say the annotation is of type `source` for brevity), and it should also have some value assigned to properties `sos:aggr_prop` (the name of the property holding the value used to aggregate other annotations as described below), `sos:step` (an incremental value used to refresh information), `sos:r_diff` (diffusion rate) and `sos:r_ctx` (contextualisation rate). This rules fires at diffusion rate `?R`: according to CTMC semantics, this means that as soon as a matching set of reactant annotations is found, actual application of the rule is delayed of a time $t$ randomly drawn according to a negative exponential distribution of probability with average value $1/$`?R`—namely, it is fired at frequency `?R`. The effect of firing this rule is twofold: it increases the value of property `sos:step` of the source annotation, and creates (by cloning the source) a new annotation `?GRAD` of type `sos:diff` and `sos:aggr` (original type `sos:source` is removed), and declaring distance $0$ and orientation `here` with respect to the source—these properties will be update as the annotation is spread around. Note that the right-hand side of a rule mentions only changes in annotations, without any need of repeating information provided in the left-hand side which has not changed: this allows a simpler structuring of rules with respect to more syntactic approaches like the one in [14].

Rule [DIFF] is used to diffuse a cloned version `?GRAD1` of a gradient annotation `?GRAD` in one neighbour. The gradient annotation should be of type `diff`, declare distance `?D` from the source and have diffusion rate `?R`. This rule has as further reactant a neighbour annotation, from which information about a neighbour `?L` with orientation `?O` and distance `?D2` can be extracted. This rule leaves the reactants unchanged, but creates a clone of `?GRAD` with type `ctx` instead of `diff`, to be relocated at `?L` and indicating orientation (with respect to originating location) `?O` and distance `?D+?D2`. Note that continuous application of this rule at rate `?R` makes copies of the gradient annotation to be created and diffused in all neighbours.

An annotation is relocated with type `ctx`, which forbids further application of rule [DIFF]. This is because we first need to aggregate all incoming annotations before a new diffusion is scheduled. To do so, rule [CTX] defers change of type from `ctx` to `diff`, which happens at rate `?RC` (contextualisation rate).

One form of aggregation is due to rule [YOUNGEST], used to refresh gradients with new information. It takes two annotations such that the values associated to the aggregation property `?P` are the same, and retains the one with bigger `sos:step`. The idea is that `sos:aggr_prop` holds the name of a property `?P` which is expected to contain some value(s) that should be identical in two annotations for them to be aggregated. For instance, it could be the unique id of a gradient source, so that we do not aggregate annotations coming from difference sources—but more involved situations can be programmed as exemplified in next section.

Rule [SHORTEST] is similar: it takes two annotations with same step (hence coming from the same gradient annotation as generated by rule [PUMP]), and retains the one with shortest distance only if, again, they have same content of aggregation property.

Finally, rule [DECAY] is used to remove an annotation at a given decay rate `?RD`. This is useful as a cleanup mechanism in the case a gradient source is removed from the system—though it plays no crucial role in this paper and will be neglected in next discussions.

For the above rules to properly work we need to synthesise a well-structured source annotation, featuring all required properties and proper values for rates—namely, diffusion rate is to be chosen to keep the system sufficiently reactive to changes without bloating the system with undesired messages, and contextualisation rate should be significantly higher than diffusion rate (at least one order of magnitude). For instance we could use the following source annotation:

```
:id314 mid:#loc :loc117;  sos:type sos:source;
       sos:step "0";      sos:sourceid "341AB2"
       sos:aggr_prop sos:sourceid;
       sos:r_diff "10";   sos:r_ctx "100"
```

### B. Rules for Gradient Composition

According to the self-composition prototype solution outlined above, we now explain how it can be operationally supported in the framework explained in Section III, particularly for the example of gradient composition with crowd level. As an assumption, we formalise recommender agent behaviours with eco-laws, as if they operate via eco-laws they inject in the node. These eco-laws specify composition patterns.

**Transition Rules for Gradients**

```
[PUMP]: An annotation of type source continuously injects the initial gradient annotation
?SOURCE sos:type sos:source; sos:aggr_prop ?P; sos:step ?T; sos:r_diff ?R; sos:r_ctx ?RC
--?R-->
?SOURCE sos:step =(?T+1) + ?GRAD(?GRAD clones ?SOURCE) sos:type -sos:source sos:diff sos:aggr; sos:dist "0"; sos:orientation "here"

[DIFF] A gradient annotation is cloned in a neighbour, with distance increased and updated orientation
?GRAD sos:type sos:diff; sos:dist ?D; sos:r_diff ?R + ?NEIGH mid:type mid:#neigh; mid:remote ?L; mid:orientation ?O; mid:distance ?D2
--?R-->
?GRAD + ?NEIGH + ?GRAD1(?GRAD1 clones ?GRAD)  sos:type -sos:diff sos:ctx; sos:dist =(?D+?D2); sos:orientation =?O; mid:#loc ?L

[CTX] A contextualising annotation is transformed back into an annotation to be diffused
?GRAD sos:type sos:ctx; sos:r_ctx ?RC
--?RC->
?GRAD sos:type sos:-ctx sos:diff;

[YOUNGEST]  Of two annotations to be aggregated based on property ?P, the one with newest information is kept
?ANN1 sos:type sos:aggr; sos:aggr_prop ?P; ?P =[?C]; sos:step ?T1 +
?ANN2 sos:type sos:aggr; sos:aggr_prop ?P2; ?P2 =[?C]; sos:step ?T2(?T2<?T1)
--->
?ANN1

[SHORTEST]  Of two annotations to be aggregated based on property ?P, the one with shortest distance from source is  kept
?ANN1 sos:type sos:aggr; sos:aggr_prop ?P; ?P =[?C]; sos:dist ?D1; sos:step ?T  +
?ANN2 sos:type sos:aggr; sos:aggr_prop ?P2; ?P2 =[?C]; sos:dist ?D2(?D2>=?D1); sos:step ?T
--->
?ANN1

[DECAY] An annotation decays
?GRA sos:type sos:diff; sos:r_dec ?RD
--?RD->
0
```

Fig. 3. Rules for gradients

**Eco-laws for gradient composition**

```
[COMPOSITION] The gradient source is composed with the crowd service
?SOURCE sos:type sos:source; scm:satisfaction ?S + ?CROWD scm:type crowd; crowd:level ?CL
--->
?SOURCE + ?CSOURCE(?CSOURCE clones ?SOURCE) scm:property sos:dist; scm:parameters scm:crowd_op  ?CF;  scm:crowd_op  ?CF*?CL

[CONTEXTUALISATION] If sensors perceive crowd, the gradient distance is augmented
?GRAD sos:type sos ctx; sos:dist ?D;  scm:property sos:dist;
        scm:parameters scm:crowd_op scm:crowd_factor;  scm:crowd_factor ?CF; scm:crowd_op  ?CF*?CL +
?CROWD scm:type crowd;  crowd:level ?CL
--->
?CROWD + ?GRAD  sos:type -sos:ctx sos:diff; sos:dist =(?D+?CF*?CL)

[FEEDBACK] Feedbacks are used to update the satisfaction values
?FEEDBACK scm:parameters scm:crowd_op; scm:feedback  scm:velocity; scm:velocity ?V +
?GRAD scm:satisfaction ?S; scm:parameters scm:crowd_op
--->
?GRAD scm:satisfaction =(?S+?V)

[EVAPORATION] The gradient satisfaction value gets decreased
?GRAD scm:satisfaction ?S; scm:factor_ev ?FE; scm:r_ev ?RE
--?RE->
?GRAD scm:satisfaction =(?FE*?S)

[DECAY] If the gradient satisfaction value becomes zero that composition is removed
?GRAD scm:satisfaction "0";
--->
```

Fig. 4. Additional rules for composition of gradients.

Rule [COMPOSITION] in Figure 4, is in charge of creating the source of the composite gradient. The new source owns the same set of properties of the basic one, as soon as it refers to the same PoI, but it also defines which properties are to be modified as an effect of the composition – `sos:dist` –, and which are composition parameters – `scm:parameters` – , *i.e.* function – `scm:crowd_op` – to be used for modifying the distance value, and coefficient —`scm:crowd_factor`. In this preliminary example we adopt a simple linear function of the crowd level perceived by sensors in each location. It increases the distance of those paths crossing crowded area, if the coefficient has an opportune value. Rule [CONTEXTUALISATION] applies the composition specified in `scm:parameters`, in each location of the system. Rule [FEEDBACK] is used to modify the satisfaction value of gradient, taking into account the velocity of the path chosen. It can be computed by the system itself, or provided by the user. The value of `scm:satisfaction` gets increased by `scm:velocity` value.

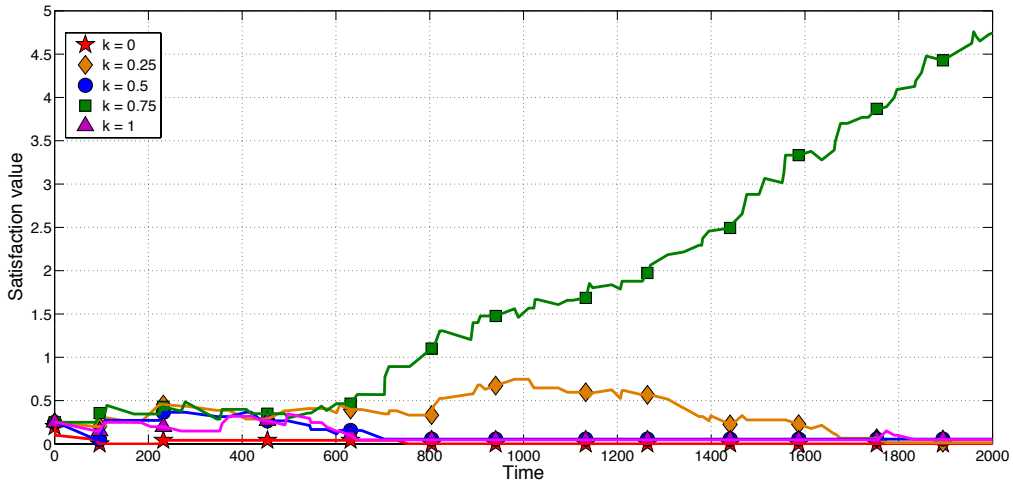


Fig. 5. Satisfaction values for different compositions changing over time.

Rules [EVAPORATION] and [DECAY] are finally used to evaporate the `scm:satisfaction` value according to an evaporation factor `scm:factor_ev ?FE` in the range of [0,1] and to remove the gradient composition in case of a `scm:satisfaction` value equal to zero, or negative, ensuring that only those compositions that compute fast paths will survive.

## VI. Towards simulation of gradient self-compositions

We checked the correctness of the proposed self-composition solution by simulation, conducted using ALCHEMIST [14], a prototype simulator extending the typical engine of a stochastic simulator for chemical reactions with the ability of expressing structured reactions (namely, where chemicals can have a tuple-like structure and reactions apply by matching) and of structuring the system as a mobile networked set of nodes. Apart from performance issues – ALCHEMIST scales better than other simulators like Repast [13] due to optimised data structures used to schedule chemical-like reactions [10] – ALCHEMIST simplifies the task of producing correct simulations since there is a small abstraction gap between simulation code and the proposed rule language.

Results presented show early experiments on gradient composition with crowd level, where people dynamically enter in the system and begin to move towards the target ascending one of the different compositions available. Movement velocity is constant in the system, except for crowded areas, where it decreases according to the crowd level perceived: higher crowd, slower velocity. In these early experiments, parameter $k$ in function $\delta$ is not computed dynamically through an evolutionary algorithm but composition agents define a set of possible values. A new composition corresponds at each value. Users chose one gradient randomly but considering also the associated satisfaction value that measures the velocity of the best path each gradient proposes. This feedback is computed by each user once reached the target, given the length of the path and the distance walked.

The goal is to observe that, satisfaction values, initially the same for each composition, dynamically change according to feedbacks from the users, and only best compositions survive. In Figure 5 such process is shown for different values of parameter $k$. The composition that wins is the one with $k = 0.75$, while the other compositions decay.

## VII. Conclusion

In this paper we propose a prototype solution for self-composition of services in pervasive systems. An illustrative model, framed in the pervasive ecosystem framework, is given. It reproduces the composition of gradient service with services, distributed in the system, providing crowd level. Early experiments have been performed for validating the approach, and even though only a little set of compositions have been considered, first results are promising.

Future works are devoted to: *(i)* introduce evolutionary techniques for varying parameter $k$; *(i)* consider gradient composition with other services, such the ones already mentioned previously in Section IV-A; *(iii)* generalise the approach towards self-composition of all the possible services available in a pervasive system.

## Acknowledgment

This work has been supported by the EU-FP7-FET Proactive project SAPERE Self-aware Pervasive Service Ecosystems, under contract no.256873

## References

[1] Notation3 (n3): A readable rdf syntax. http://www.w3.org/TeamSubmission/n3/, 2011.
[2] RDF primer. http://www.w3.org/TR/rdf-primer/, 2011.

[3] J. Beal. Flexible self-healing gradients. In S. Y. Shin and S. Ossowski, editors, *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*, pages 1197–1201. ACM, 2009.

[4] J. Beal and J. Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, 2006.

[5] M. Beek, A. Bucchiarone, and S. Gnesi. A survey on service composition approaches: From industrial standards to formal methods. In *Technical Report 2006TR-15, Istituto*, pages 15–20. IEEE CS Press, 2006.

[6] A. A. Berryman. The origins and evolution of predator-prey theory. *Ecology*, 73(5):1530–1535, October 1992.

[7] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1069–1075, New York, NY, USA, 2005. ACM.

[8] J. L. Fernandez-Marquez, G. Di Marzo Serugendo, S. Montagna, M. Viroli, and J. L. Arcos. Description and composition of bio-inspired design patterns: a complete overview. *Natural Computing*, May 2012. Online First.

[9] J. L. Fernandez-Marquez, G. Di Marzo Serurendo, and S. Montagna. Bio-core: Bio-inspired self-organising mechanisms core. (to appear). In *6th International ICST Conference on Bio-Inspired Models of Network, Information, and Computing Systems (Submitted)*, 2011.

[10] M. A. Gibson and J. Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *J. Phys. Chem. A*, 104:1876–1889, 2000.

[11] R. Kazman and H.-M. Chen. The metropolis model a new logic for development of crowdsourced systems. *Commun. ACM*, 52:76–84, July 2009.

[12] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. Softw. Eng. Methodol.*, 18(4):1–56, 2009.

[13] M. J. North, T. R. Howe, N. T. Collier, and J. R. Vos. A declarative model assembly infrastructure for verification and validation. In *Advancing Social Simulation: The First World Congress*, pages 129–140. Springer Japan, 2007.

[14] D. Pianini, S. Montagna, and M. Viroli. A chemical inspired simulation framework for pervasive services ecosystems. In M. Ganzha, L. Maciaszek, and M. Paprzycki, editors, *Proceedings of the Federated Conference on Computer Science and Information Systems*, pages 675–682, Szczecin, Poland, 18-21 September 2011. IEEE Computer Society Press.

[15] N. Salazar, J. A. Rodriguez-Aguilar, and J. L. Arcos. Robust coordination in large convention spaces. *AI Communications*, 23(4):357–372, 2010.

[16] A. Seaborne and S. Harris. SPARQL 1.1 query. W3C working draft, W3C, Oct. 2009. http://www.w3.org/TR/2009/WD-sparql11-query-20091022/.

[17] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Web Semant.*, 5:51–53, June 2007.

[18] M. Viroli. A self-organising approach to competition and composition of pervasive services. Technical Report SAPERE TR.WP2.2011.4, 2011.

[19] M. Viroli, E. Nardini, G. Castelli, M. Mamei, and F. Zambonelli. A coordination approach to adaptive pervasive service ecosystems. In *IEEE International Conferences on Self-Adaptive and Self-Organizing Systems – Workskop AWARE 2011*, 2011.

[20] M. Viroli, D. Pianini, S. Montagna, and G. Stevenson. Pervasive ecosystems: a coordination model based on semantic chemistry. In S. Ossowski, P. Lecca, C.-C. Hung, and J. Hong, editors, *27th Annual ACM Symposium on Applied Computing (SAC 2012)*, Riva del Garda, TN, Italy, 26-30 March 2012. ACM.

[21] Z. Wu, A. Ranabahu, K. Gomadam, A. Sheth, and J. Miller. Automatic composition of semantic web services using process and data mediation. In *Proc. of the 9th Intl. Conf. on Enterprise Information Systems*, pages 453–461, 2007.

[22] F. Zambonelli and M. Viroli. A survey on nature-inspired metaphors for pervasive service ecosystems. *International Journal of Pervasive Computing and Communications*, 7(3):186–204, 2011.