UNIVERSITÉ DE GENÈVE

Archive ouverte UNIGE
https://archive-ouverte.unige.ch

Rapport technique 2013

Open Access

This version of the publication is provided by the author(s) and made available in accordance with the copyright holder(s).

Self-composition of services with chemical reactions

De Angelis, Francesco; Di Marzo Serugendo, Giovanna; Fernandez Marquez, Jose Luis

How to cite

DE ANGELIS, Francesco, DI MARZO SERUGENDO, Giovanna, FERNANDEZ MARQUEZ, Jose Luis. Self-composition of services with chemical reactions. 2013

This publication URL: https://archive-ouverte.unige.ch/unige:32649

© This document is protected by copyright. Please refer to copyright holder(s) for terms of use.

Self-composition of services with chemical reactions

Francesco De Angelis, Giovanna di Marzo, Jose Luis Fernandez-Marquez
francesco.deangelis@unige.ch, JoseLuis.Fernandez@unige.ch, giovanna.dimarzo@unige.ch
University of Geneva, ISS
Battelle, Batiment A, Route de Drize 7, CH-1227
Carouge SWITZERLAND

***Abstract*—Service-oriented programming has dramatically changed the way software applications are developed. Automatic composition of services reduces human interaction on choosing the set of services that satisfy a specific request. Automatic composition has mainly been done in a centralised way with the composition specified at design time or in a decentralised manner through multi-agent systems planing. These approaches present several limitations when used in dynamically changing environments like pervasive computing scenarios. Very few proposals investigate the design of the composition arising as a result of an ongoing self-organising process that permanently adapts to context and environmental changes. In this paper we define three chemically inspired approaches for self-composition of services operating on top of a shared tuple space. We show how tuple spaces can be exploited to design spontaneous and emergent compositions that deal with context information and a dynamic set of available services. Then we prove that our algorithms can be generalised to produce compositions of services with very common features. and show how our algorithms can be easily implemented to provide self-composition of actual Web services.**

***Keywords*-Self-composition, chemical reactions, services, context-awareness, dynamic environment**

I. Introduction

A software service represents a functionality that can be provided on demand in order to create higher level software artifacts, 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.

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, or AI planning techniques to generate process automatically based on the specification of a problem. Their basic goal is to analyse a description of the services to compose and compute a composition satisfying structural, behaviour and 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 or context-awareness to environmental conditions or to the appearance or disappearance of services.

This paper discusses a model and several approaches for providing self-composition of services based on chemical reactions occurring in an active shared tuple space. Descriptions of services, their queries and their answers are injected in the shared tuple space under the form of tuples. Chemical reactions active in the tuple space act on the tuples causing them to link with each other, and eventually leading to zero, one or more compositions of services that can answer the query (this depends on the actual services available). Context-awareness is present in the system in different manners: (1) available services are advertised in the system through a tuple that describes them (functional or non-functional information). If they disappear, their respective tuple disappears as well; (2) the self-composition process takes into account contextual information present in the tuple space, such as availability or quality of service of other services, partially built composition of services, or environmental related information provided by sensors or services themselves.

We also show how this model has been implemented on an actual middleware and how actual Web services can be wrapped and used to participate into self-compositions of services. We also discuss complexity related issues and show that both at the level of services and at the level of chemical reactions, the approach is linear showing the scalability of the approach.

Section II discusses related works. Section III presents the system model, while Section IV presents our three approaches for self-composition based on chemical reactions. Section V shows an actual implementation involving Web Services, and Section VI discusses conclusions and perspectives.

II. Related work

On-the-fly service or components composition approaches span from workflow based approaches or AI Planning [1] to more dynamic ones where a central coordinator organises the assembly of components [2], to more decentralised ones,

where devices, providing services, listen to channels for messages or queries to process and a distributed conflict resolution system solves competing issues among similar services [3].

Multi-agent systems approaches to self-composition usually involve planification, where agents reason on their respective services and the user's needs [4]. In this area, recent work on self-composition of methods fragments brings a solution based on cooperative agents, each representing a fragment and participating to the design of the fragments composition [5].

Approaches specifically involving chemical reactions for self-composition include the following. In the field of industrial robotics, Frei et al. [6] propose the use of chemical reactions to build self-organising assembly systems that participate in their own design by spontaneously organising themselves in the shop floor layout in response to the arrival of a product order. This approach shares similarities to the one presented in this paper, since chemical reactions act on a shared space containing robotic modules capabilities and needs, and generic assembly instructions. However, this approach does not use an actual middleware and the actual chemical process is performed through specifications logic rewriting in MAUDE.

Di Napoli et al. [7] show how a specified workflow can be instantiated using chemical reactions. In this approach, chemical reactions are used for "binding" the services to the pre-specified workflow. This differs from our approach, where there is no predefined workflow. The actual composition design emerges from the chemical reactions at work.

Finally, Viroli et al. [8] propose an approach inspired by chemical reactions combined with the notion of competition among services. This approach addresses services that span multiple hosts and permanently push data in a tuple space, rather than services activated in response to queries.

III. Reference model

This section presents the reference model used to describe our approach for self-composition of services.

The model we are dealing with is an abstraction of the active tuple space model SAPERE [9], [10] inspired by chemical reactions; here the system is composed of four entities: *tuples*, *chemical reactions*, *agents* and *services/applications* (Figure 1).

Tuples: A tuple $\bar{T}$ is a vector of elements $\bar{P}_i = (\text{“}N_i\text{”}, V_i)$ called *properties*, thus:

$$\bar{T} = (\bar{P}_1, \bar{P}_2, \ldots, \bar{P}_n)$$

where "N_i" is the name of the property and V_i is a value. Tuples are passive entities located in a shared container named *tuple space*. Tuples are dynamically updated by the agents and the tuple space. They represent both services and context information.

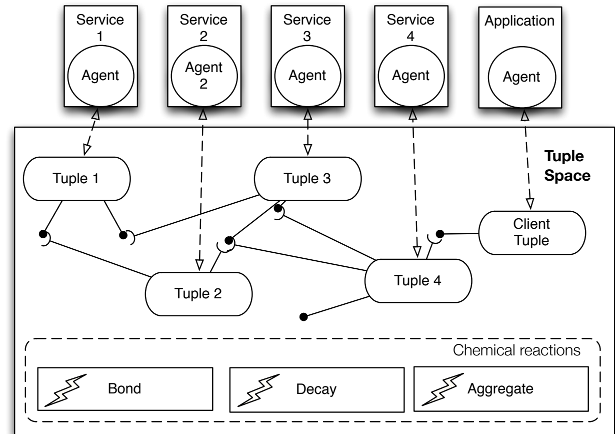

Figure 1: Reference model. Dotted lines represent interaction between agents and their tuple in the tuple space; Solid lines represent interactions among tuples.

Chemical reaction: If $\mathcal{S}$ is the set of tuples, then a chemical reaction r is a function over the power set $\mathcal{P}(\mathcal{S})$:

$$r : \mathcal{P}(\mathcal{S}) \rightarrow \mathcal{P}(\mathcal{S})$$

Chemical reactions provide the system with an automatic way for tuples interactions, combining and updating them. When chemical reactions are fired notifications are delivered to agent associated with tuples which are modified in the space.

Agents: An agent is an external active entity represented in the tuple space by a tuple. Every agent interacts with the tuple space updating-deleting tuples and receiving a notification each time an interaction is performed on its tuple by a chemical reaction. Agents are part of applications and services.

Services/Applications: When an application or a service wants to request or provide information, it creates an agent that it uses as an interface to exchange data within the tuple space. In order to invoke a service, applications create agents that insert tuples containing input parameters for a request. Such tuples are then managed by service agents which are in charge of reading input parameters and passing them to the service. When results are computed, service agents will make them available to application agents by inserting them as a new tuple in the space.

A. Chemical reactions and interactions

Chemical reactions fire automatically in the tuple space and operate on tuples containing special data named *operators*. We define three basic chemical reactions: *Bond*, *Decay* and *Aggregate*.

Bond. A bond is a kind of relationship between two properties of two different tuples and it is the main way to realise

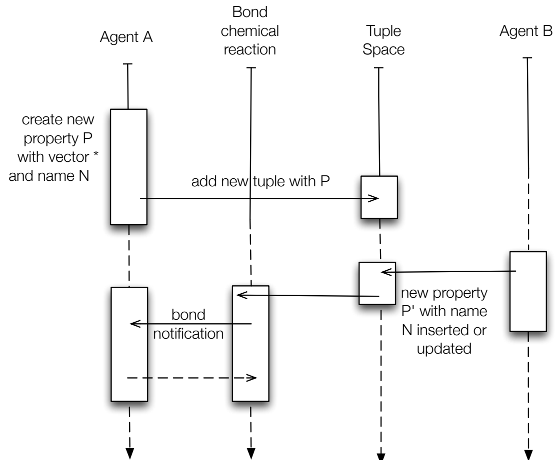

Figure 2: The bond chemical reaction.

an interaction among tuples.
When an agent A, associated with a tuple $\bar{T}_A$, wants its tuple to react in presence of another tuple containing a property $\bar{P}_b = (\text{“}N_b\text{”}, V_b)$, it adds to $\bar{T}_A$ a property filled up with the "*" operator:

$$\bar{P}a = (\text{“}N_b\text{”}, V_a) \text{ where } V_a = \text{“} * \text{”}$$

From this moment on (Figure 2), the bond chemical reaction takes care to deliver to A a notification each time a tuple with a property named "N_b" appears in a tuple of the space, changes its content or is removed from the tuple space. A is then able to read the actual value V_b and use it to manage additional computations, for example passing V_b as input parameter to a service, requesting its invocation.
Decay. It provides a mechanism to free resources by removing tuples from the space. The agent A communicates that its tuple $\bar{T}_A$ has to be removed after t_{rem} units of time by adding to it a property using the *"Decay"* operator:

$$\bar{P}_d = (\text{“}decay\text{”}, V_{rem}) \text{ where } V_{rem} = t_{rem}$$

After the creation of $\bar{P}_d$, the decay chemical reaction is in charge of delivering to A a notification after a temporal period of length t_{rem}, before proceeding with removing $\bar{T}_A$ from the space (Figure 3).
Aggregate. It is in charge of merging together several indexed tuples, producing a new one filled with synthesised data:

$$\bar{P}_{i_R} = (\text{“}aggregator\text{”}, V_{i_R})$$

This reaction is carried on by using aggregation specifications contained in V_{i_R}, that are discussed in details in section IV-D.

By using the model and the interactions previously defined we are able to define three approaches to perform self-composition of services. As said above, we assume that each

Algorithm 1: Invoking all services - answer: Service agent A_i

`OnServiceInitialization():`
insert $\bar{T}_i = (\bar{P}_i)$ with $\bar{P}'_i = (\text{“}x_i\text{”}, V_i)$, $V_i = \text{“} * \text{”}$;
`OnServiceNotification():`
x_i = read input value V_i from tuple;
calculate $y_i = f_i(x_i)$;
insert $\bar{T}_{ri} = (\bar{P}'_i)$ with $\bar{P}'_i = (\text{“}y_i\text{”}, V'_i)$,$V'_i = y_i$;

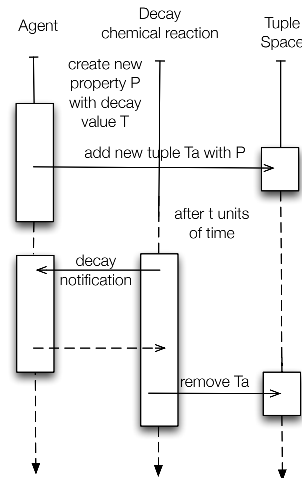

Figure 3: The decay chemical reaction.

existing service S_i has an agent A_i related to it, with a tuple $\bar{T}_i$ created in the tuple space that represents it. We assume that every service takes as input just one parameter x_i and generates a single value y_i (or a structured value that has to be consider atomic) i.e. the service can be mapped as a scalar function: $y_i = f_i(x_i)$. We are showing how to remove this assumption in the section IV-D, where we generalise our proposed algorithms for services which accept a greater number of input parameters, i.e. for services mapped as vector-valued functions: $\bar{y}_i = f_i(\bar{x}_i)$. For a request to the service S_i, the query of an application agent is represented by a tuple containing x_i, i.e. the specified input parameter; similarly, when a service produces the result for an invocation, the associated service agent inserts a new tuple in the space containing the response y_i.

IV. Self-composition algorithms

We present now our three approaches for self-composition.

A. Executing all services

Our first approach consists in defining a strategy to trigger a global process of callings all services related to the query, thus leading up to the production of the result requested by

an application.
Input/Output: During its creation, each service agent A_i instantiates the following tuple:

$$\bar{T}_i = (\bar{P}_i) \text{ where } \bar{P}_i = (\text{“}x_i\text{”}, V_i),\ V_i = \text{“}*\text{”}$$

When the service agent receives the notification of a new bond with a tuple containing a property filled with the input parameter, it reads the value x_i from the second tuple, it calculates y_i and it generates a new tuple $\bar{T}_{ri}$:

$$\bar{T}_{ri} = (\bar{P}'_i) \text{ where } \bar{P}'_i = (\text{“}y_i\text{”}, V'_i),\ V'_i = y_i$$

Query: A generic application that wants to get a value of type “b” by providing a value of type “a” creates an agent A_c that simply inserts a tuple $\bar{T}_c$:

$$\bar{T}_c = (\bar{P}_{c_1}\bar{P}_{c_2}) \text{ where } \bar{P}_{c_1} = (\text{“}a\text{”}, a),\ \bar{P}_{c_2} = (\text{“}b\text{”}, \text{“}*\text{”})$$

Composition: When $\bar{T}_c$ is inserted in the tuple space, the bond chemical reaction delivers the first notification to a service S_1 accepting a and producing y_1. If the latter value is the input for a second service S_2 then the bond will deliver a further notification event to its agent A_2, which generates a second value of type “y_2”. This process, automatically carried out by the system, continues for all $n \geq 1$ services involved and it stops when A_c receives the notification for a bond with a tuple containing the requested result. At this point, the application agent can read the value of type “b” generated through the composition:

Algorithm 2: Invoking all services - query for values of type “b” provided a: application agent A_c

---

`OnApplicationInitialization():`
 insert $\bar{T}_c = (\bar{P}_{c_1}, \bar{P}_{c_2})$ where $\bar{P}_{c_1} = (\text{“}a\text{”}, a)$
 and $\bar{P}_{c_2} = (\text{“}b\text{”}, \text{“}*\text{”})$;
`OnApplicationResponse():`
 b = read input value from tuple;

$$b = (f_n \circ f_{n-1} \circ \cdots \circ f_1)(a) \text{ with } n \geq 1 \qquad (1)$$

The proposed approach is summarised in Algorithm 1 and in Algorithm 2. Depending on how responses trigger the invocation of services, this process may potentially involve all the services of the space, even those which do not contribute to generating b; an instance of tuple space at the end of the answer generation is depicted in Figure 4.

B. Designing all compositions

The first approach may not stop in case of services whose outputs generate a cyclic invocation of services, so additional computations have to be executed to avoid loops during the generation of a result. Moreover, there may be more than one composition that leads at generating the result of type “b” starting from a value “a” (i.e. several equations

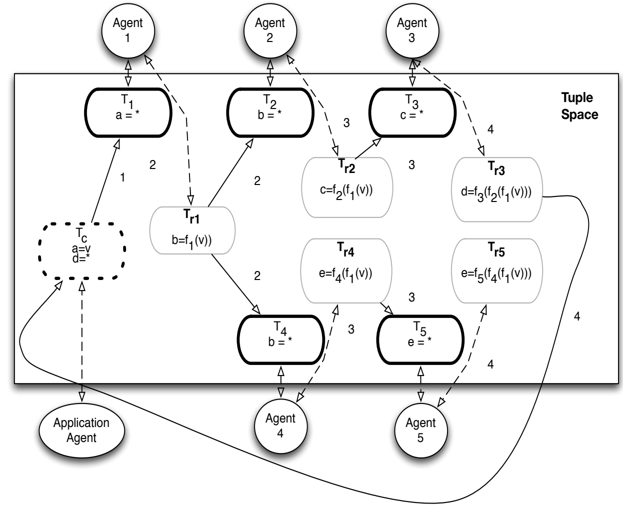

Figure 4: Starting all services: generation of the requested value. Continuos lines point out tuple triggering bonds with other tuples in the space. Edge labels define instants of time when tuples are created/activated. A query for a value of type “a” (tuple $\bar{T}_c$) triggers the execution of all the services but only tuple T_{r3} is really used to carry on the result of composition. In bold lines tuples related to services that are activated in the process.

like Eq.(1)), so having the chance of selecting the sequence of services to invoke could be useful for several reasons, for example to satisfy *level service agreements* contracted among applications and services.

Our second approach, summarised in Algorithm 3 and Algorithm 4, is structured in two phases: the *discovery process* and the *request process*. The former aims to discover one or more candidate sequences of services invocation $\bar{S}_{seq}$ providing the requested composition, the latter leads to the actual execution of $\bar{S}_{seq}$, invoking the specified services in the right order. During the *discovery process*, every service agent A_i keeps a property $\bar{P}_{i_l} = (\text{“}neighbours\text{”}, V_{i_l})$ updated with the list of all the agents A_z able to provide the value x_i as a result of a computation. We call them “neighbours” because we refer to them as agents able to provide something that can be taken as input by A_i during a composition. For the sake of clarity, from now on, V_{i_l} is treated as a vector. For such an agents, A_i also keeps in its memory a local copy of each V_{j_l} within a set named $\mathcal{L}_i$.
Input/Output: When the service agent A_i is created, it inserts the tuple $\bar{T}_i$ in the space:

$$\begin{aligned} &\bar{T}_i = (\bar{P}_{i_1}, \bar{P}_{i_2}, \bar{P}_{i_3}, \bar{P}_{i_l}) \text{ where } \bar{P}_{i_1} = (\text{“_}x_i\text{”}, \text{“}*\text{”}) \\ &\bar{P}_{i_2} = (\text{“_}y_i\text{”}, \text{“}A_i\text{”}) \quad \bar{P}_{i_3} = (\text{“}x_i\text{”}, \text{“}*\text{”}) \\ &\bar{P}_{i_l} = (\text{“}neighbours\text{”}, \text{“”}) \end{aligned} \qquad (2)$$

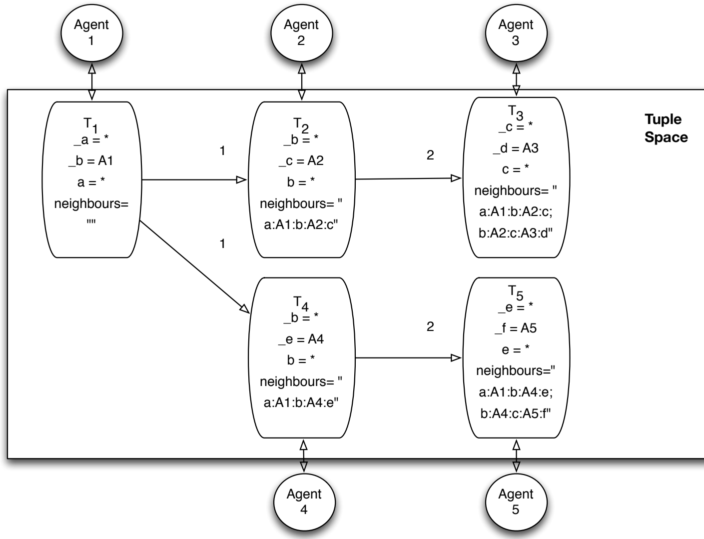

Figure 5: An instance of the tuple space at the end of the discovery process. Solid lines point out bonds between tuples of services during the reading of their property "neighbour". Edge labels define instants of time when tuples are created/activated.

The first property is used to inform other agents A_j whose input parameter "x_j" equals the output parameter of A_i that A_i provides in output a result of type "y_i" that they can take as input for a computation. Similarly, $\bar{P}_{i_2}$ is used to let A_i discover agents A_z with "y_z" = "x_i", i.e. services that provide values that can be taken as input by A_i. The character "_v_i" before the name of a variable v_i is a notation used to highlight the purpose of the bond we are dealing with: we do not want to read a value of type v_i, we are just interested in discovering which agent is able to provide that parameter. $\bar{P}_{i_3}$ is used by A_i to be bond with requests for services accepting inputs of type 'x_i".

Composition Design: At the beginning, $\bar{P}_{i_l}$ is empty; each time the agent A_i receives a notification of a bond with a tuple $\bar{T}_z$, it looks for property $\bar{P}_{z_l}$. If it is existing and if the content of $\bar{P}_{z_l}$ has changed since last notification it reads all the properties of $\bar{T}_z$, it merges its own V_{i_l} with V_{z_l} and finally it adds to V_{i_l} (if not contained yet) the following value:

$$\text{“}A_i : y_i : x_z : A_z : y_z\text{”} \tag{3}$$

This 5-tuple is stating that if a service or an application passes a value of type "x_z" to agent "A_z" then it produces a value of type "y_z" accepted as input by "A_i", which finally generates an output of type "y_i". So given A_i, the invariant property that holds during the update of V_{i_l} is the following:

$$\forall e = \text{“}t : u : x_w : A_w : z\text{”} \in V_{i_l}\ \exists y_i, A_{h_1}, \ldots, A_{h_{n-1}} : \\ y_i = (f_i \circ f_{h_1} \circ \cdots \circ f_{h_{n-1}} \circ f_w)(x_w) \qquad \forall t, u, z$$

with $A_w, A_{h_1}, \ldots, A_{h_{n-1}}$ service agents. In other words, V_{i_l} contains the list of all the agents that triggers the execution of A_i when they produce a result. At the end of the discovery process each service agent A_i contributes to defining a graph of "relations of tuples", which looks like the one depicted in Figure 5. If that graph does not contain loops, almost every A_i is going to have just a partial view of it, saving memory resources.

Query: When an application agent A_c wants to obtain a value of type "b" starting from a type "a" (Algorithm 3), it first inserts a tuple $\bar{T}_c$:

$$\bar{T}_c = (\bar{P}_c) \text{ where } \bar{P}_c = (\text{“_}b\text{”}, \text{“}*\text{”}))$$

A_c is going to receive as many bond notifications as the number of services providing "b"; this happens because of property $\bar{P}_{i_2}$ in Eq.(2). At this point A_c reads the tuple $\bar{T}_i$, it extracts V_{i_l} and it finds a sequence $\bar{S}_{seq} = (A_z, \ldots, A_i)$ of services invocations from each service agent A_z accepting "a" (i.e. "x_z" = "a") to A_i; this is equivalent to solve the shortest-path problem related to finding a path from a to b in the graph described above. $\bar{S}_{seq}$ acts like an activator for services invocation, in the sense that it contains the list of services that have to be invoked to provide the the answer for a query. By knowing $\bar{S}_{seq}$, A_c starts the *request process* by inserting a new tuple $\bar{T}'_c$:

$$\bar{T}'_c = (\bar{P}_{c_1}, \bar{P}_{c_2}, \bar{P}_{c_3}) \text{ where } \bar{P}_{c_1} = (\text{“}b\text{”}, \text{“}*\text{”}), \\ \bar{P}_{c_2} = (\text{“}a\text{”}, a)\bar{P}_{c_3} = (\text{“}activator\text{”}, \bar{S}_{seq})$$

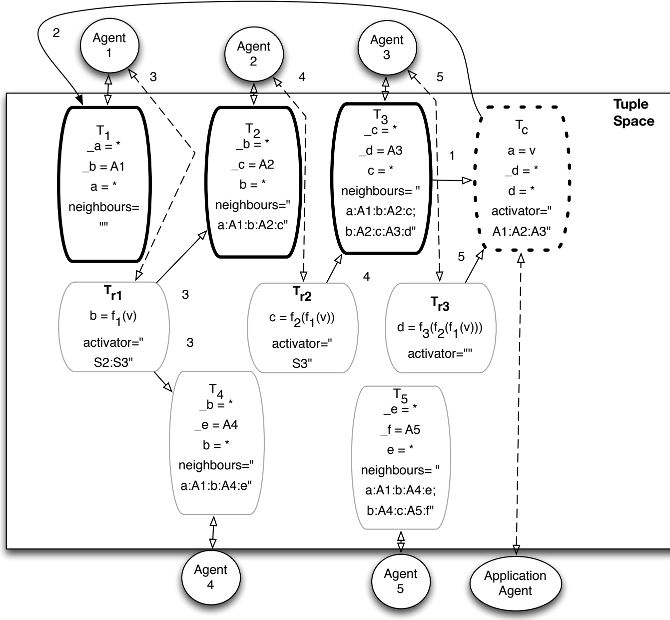

Figure 6: An instance of the tuple space at the end of the request process. Solid lines point out tuple triggering bonds with other tuples in the space. Edge labels define instants of time when tuples are created/activated. Bold lines tuples relate to services activated during the composition. A request for a value of type "d" activates a sequence of executions that do not involve services 4 and 5.

Composition Activation: Each time a service A_i receives a notification because of a bond related to the property $\bar{P}_{i_3}$ in Eq.(2), it reads $\bar{S}_{seq}$ from the tuple and it checks if its identificator is present as first element of the sequence. If not, it discards the notification. Otherwise, it reads x_i, it calculates $y_i = f_i(x_i)$, it removes the first element of $\bar{S}_{seq}$ producing $\bar{S}'_{seq}$ and it finally inserts a new tuple $\bar{T}_r$:

$$\bar{T}_r = (\bar{P}_{r_1}, \bar{P}_{r_2}) \text{ where } \bar{P}_{r_1} = (\text{"}y_i\text{"}, y_i)$$
$$\bar{P}_{r_2} = (\text{"}activator\text{"}, \bar{S}'_{seq})$$

The request process stops when the last agent of the sequence generates b, triggering a notification bond for A_c which reads the tuple obtaining the final result. An instance of the request process is shown in Figure 6.

Services Removal: We conclude this section showing how to manage the removal of a service. When a service agent A_z is removed, its tuple $\bar{T}_z$ has been removed as well from the tuple space; in this case, every A_i having a bond with A_z receives a delete notification triggered by the bond chemical reaction. A_i identifies what agent A_z has disappeared, it removes its local copy of $\bar{P}_{z_l}$ and then it updates $\bar{P}_{i_l}$ as follow:

$$H = \{e = \text{"}s_1 : s_2 : s_3 : A_j : s_4\text{"} \textit{ in } V_{i_l} : V_{z_l} \in \mathcal{L}_i\}$$
$$V_{i_l} = (e_1, \ldots, e_n) \textit{ where } e_j \textit{ in } V_{z_l} \vee e_j \textit{ in } H,$$
$$V_{z_l} \in \mathcal{L}_i \;\; \textit{for any } s_1, \ldots, s_4 \; \forall j = 1, \ldots, n$$

After updating V_{i_l}, by using the bond chemical reaction, the update is going to spread all over the agents by using the same strategy used to provide the list of "neighbours" to every service agent.

C. *Reasoning on top of the compositions*

In our third approach, the process of designing the best composition of services to invoke procedure is entirely performed through an intermediate agent, named "agent proxy", It takes care to realise the discovery process - in a manner similar to our second approach - and to initialise the request process.

Query: When a generic application wants to calculate a value of type "b" starting from a value of type "a", it simply inserts a tuple $\bar{T}_c$ containing three properties: one for the input parameter, one for the output parameter and an additional property triggering a bond with the proxy agent.

Composition Design and Activation: Once the proxy agent receives a notification for a bond with an application, it

establishes a second bond with the service agent providing the result, it calculates the sequence $\bar{S}_{seq}$ and it inserts a new tuple containing a, which starts the request process as discussed in the previous section.
This third approach presents three important improvements: application agents do not deal with the discovery process anymore and the proxy agent may potentially pre-calculate all the distinct compositions of services or store them, speeding the discovery process. Moreover they may record additional information about services, which contribute to creating a global view of the features of the components within the system. The first improvement simplifies the design of an application agent, reducing its complexity. The second one decreases the average amount of time to carry out a discovery process: in case of n services with just one input parameter, the number N_c of distinct compositions of two or more services equals to $\frac{n(n-1)}{2}$; by using a cache of N_c entries it is then possible to calculate the best sequence of services just once. The third one ensures that the research of the optimal sequence can take into account supplementary metrics in addition to the number of services involved in composition.

Algorithm 3: Composition Design - query for values of type "b" given input a: application agent A_c

```
OnApplicationInitialization():
  insert T̄_c = (P̄_c) where P̄_c = ("_b", "*");
OnApplicationNotification():
  if (tuple with bond contains a result) then
  |  call OnApplicationRequestNotification();
  else
     V_{i_l} = read property "neighbours" from tuple;
     define set S_t of agents in V_{i_l} accepting type "a";
     foreach (agent A_j ∈ S_t) do
     |  calculate shortest path from A_j to A_i
     end
     S̄_seq = get minimum shortest path;
     insert tuple T̄_i = (P̄_{c1}, P̄_{c2}, P̄_{c3}) where
     P̄_{c1} = ("b", "*"), P̄_{c2} = ("a", a)
     P̄_{c3} = ("activator", S̄_seq);
  end
OnApplicationRequestNotification():
  b = read input value from tuple;
```

D. Generalization for services with $n > 1$ input parameters.

In this section we are going to show how to remove the assumption about single parameter input, thus making our approaches supporting any type of compositions, not only sequences of compositions. Let us assume we are dealing with the second strategy (the third would be approximatively the same) and let S_i be a service with input vector $\bar{x}_i = (x_{i1}, \ldots, x_{in})$, producing in output $\bar{y}_i = (y_{i1}, \ldots, y_{im})$.

Input/Output: Every service agent A_i, during its initialization, is going to insert the following tuple $\bar{T}_i$:

$\bar{T}_i = (\bar{P}_{i_{11}}, \ldots, \bar{P}_{i_{1n}}, \bar{P}_{i_{21}}, \ldots, \bar{P}_{i_{2m}}, \bar{P}_{i_{31}}, \ldots, \bar{P}_{i_{3n}}, \bar{P}_{i_l})$
where $\bar{P}_{i_{1j}} = (\text{"_}x_{ij}\text{"}, \text{"}*\text{"})\ \forall j = 1, \ldots, n$
$\bar{P}_{i_{2j}} = (\text{"_}y_{ij}\text{"}, \text{"}A_i\text{"})\ \forall j = 1, \ldots, m$
$\bar{P}_{i_{3j}} = (\text{"}x_{ij}\text{"}, \text{"}*\text{"})\ \forall j = 1, \ldots, n$ $\bar{P}_{i_l} = (\text{"}neighbours\text{"}, \text{""})$

Composition Design: The discovery process equals the one reported in section IV-B except for the kind of properties used; now each agent must export the interfaces for each input and output parameter. Properties $\bar{P}_{i_{3j}}$ are used to create bonds with tuples containing at least one input parameter: during the bond notification it is up to the agent A_i to check if a tuple contains all of the required inputs. This means that a service can take as input a subset of the elements contained in a (vector) result produced by another service. The "neighbours" property $\bar{P}_{i_l}$ now is going to contain elements of the following type:

$$\text{"}A_i : y_{i1} : \cdots : y_{im} : x_{z1} : \cdots : x_{zn} : A_z : y_{z1} : \cdots : y_{zm}\text{"}$$

which are the generalization of Eq.(3). Tuples are now indexed by using a property $\bar{P}_{i_d}$; they also define a property $\bar{P}_{i_R}$ which contains a set of aggregation specifications defining the synthesis of several tuples, allowing services to take as input vectors containing results from several service invocations. There exist two kinds of specifications; the first ones are elaborated by service agents and they are represented by strings of this type:

$$\text{"}i_{dz} : x_{z1} : \cdots : x_{zm} : i_{dk}\text{"} \quad (4)$$

which are stating that if a service agent A_i is elaborating a request triggered by a tuple with index i_{dz} containing input parameters $x_{z1}, \ldots, x_{zm}$ then the tuple containing the result has to be indexed with id i_{dk}. The second type are elaborated by the Aggregate chemical reaction and are represented by strings of this type:

$$\text{"}i_{dz} : i_{dk}\text{"} \quad (5)$$

Specifications of this kind state that all tuples with index i_{d_z} have to be merged together in a new tuple with index i_{dz}. In other terms, specifications of type (4) identify results whereas specifications of type (5) define a way to combine them in order to obtain a vector of results to pass to another service. During a computation of a service, properties $\bar{P}_{i_d}$ have to be copied in the tuple containing the result of the computation; similarly, during an aggregation, specifications for tuples being merged flow into the property $\bar{P}_{i_R}$ of the final tuple. This way, specifications can be interpreted by all the agents involved in a composition. An application agent (or a proxy agent) has to calculate the right sequence of service invocations for all the temporary results used in a composition; this is performed again by using the discovery process as mentioned before.

Query and Composition Activation: When this process

ends, the agent starts the request process creating as many tuples as the number of input parameters needed in the composition, each one containing the set of specifications that leads up to the specified composition.
For example, let us assume that we have three services represented by the following functions:

$$c = f_1(a)\ d = f_2(b)\ e = f_3(c,d)$$

We want to calculate e knowing a and b. At the beginning, the tuple space contains:

$$\begin{aligned}&\bar{T}_1 = ((\text{“_}a\text{”}, \text{“}*\text{”}), (\text{“_}c\text{”}, \text{“}A_1\text{”}), (\text{“}a\text{”}, \text{“}*\text{”}),\\&(\text{“}neighbours\text{”}, \text{“”}))\ \bar{T}_2 = ((\text{“_}b\text{”}, = \text{“}*\text{”}),\\&(\text{“_}d\text{”}, \text{“}A_2\text{”}), (\text{“}b\text{”}, (\text{“}*\text{”})), (\text{“}neighbours\text{”}, \text{“”}))\\&\bar{T}_3 = ((\text{“_}c\text{”}, \text{“}*\text{”}), (\text{“_}d\text{”}, \text{“}*\text{”}), (\text{“_}e\text{”}, \text{“}A_3\text{”}),\\&(\text{“}c\text{”}, \text{“}*\text{”}), (\text{“}neighbours\text{”}, \text{“”}), (\text{“}d\text{”}, \text{“}*\text{”}),\end{aligned} \tag{6}$$

At the end of the discovery process, by solving two distinct shortest-path problems, the application has discovered that e can be produced from $f_3(c,d)$ with $f_1(a)$ and $f_2(b)$, values calculated starting from a and b. At this point, it generates the following set of specifications:

$$\begin{aligned}V_{1_R} = V_{2_R} = \{&\text{“}i_{d1} : a : i_{d3}\text{”}, \text{“}i_{d2} : b : i_{d3}\text{”},\\&\text{“}i_{d3} : i_{d4}\text{”}, \text{“}i_{d4} : a : b : i_{d5}\text{”}\}\end{aligned}$$

then it starts the request process by inserting its main tuple:

$$\bar{T}_c = ((\text{“}e\text{”}, \text{“}*\text{”}))$$

along with two additional tuples:

$$\begin{aligned}&\bar{T}_{c1} = ((\text{“}a\text{”}, a), (\text{“}activator\text{”}, (\text{“}A_1\text{”}, \text{“}A_3\text{”})),\\&(\text{“}aggregator\text{”}, V_{1_R})), \bar{P}_{c1_d}\ \bar{P}_{c1_d} = (\text{“}id\text{”}, id_1)\\&\bar{T}_{c2} = ((\text{“}b\text{”}, a), (\text{“}activator\text{”}, (\text{“}A_2\text{”}, \text{“}A_3\text{”})),\\&(\text{“}aggregator\text{”}, V_{2_d})), \bar{P}_{c2_d}\ \bar{P}_{c2_d} = (\text{“}id\text{”}, id_2)\end{aligned}$$

When service agent A_1 generates $b = f_1(a)$ it creates $\bar{T}_{r1}$ associating it with index i_{d3}; A_2 does the same, inserting $c = f_2(b)$ in $\bar{T}_{r2}$ and setting the index i_{d4}. Finally the Aggregate chemical reaction, applies the third specification, merges the above-mentioned tuples producing:

$$\bar{T}_{c11} = ((\text{“}c\text{”}, f_1(a)), (\text{“}d\text{”}, f_2(b)), (\text{“}activator\text{”}, \text{“}A_3\text{”}))$$

It is important to note that the activator is always the same for all the tuples being merged, so there is no additional computation to update it. When $\bar{T}_{c11}$ is created, it triggers a bond with $\bar{T}_3$ and the service agent A_3 produces e in a tuple with id i_{d5}; at this point, the application agent A_c receives a notification and it reads the final result. This process provides an automatic way for parallel execution of services as soon as tuples containing their input vectors are ready in the space.
The following theorem states that any composition of $n \geq 1$ services can be defined by using specifications (4) and (5).

Theorem 1. *Let $f^{(n)}(x_1, \ldots, x_k)$ be a function of k parameters defined as composition of $n \geq 1$ nested functions (i.e. simulating $f^{(n)}$ on a machine with a pushdown-stack, the stack reaches a depth equals to n at least once). Then it exists a list $\mathcal{R}_n$ of specifications (4) and (5) which can be used to realise $f^{(n)}$.*

Proof: We proceed by induction on n.
Base $n = 1$ In this case, it exists $g(x_1, \ldots, x_k)$ such that $f^{(n)}(x_1, \ldots, x_k) = g(x_1, \ldots, x_k)$. Let assume that the j-th parameter x_j is contained into a tuple with index id_j. Then

$$\begin{aligned}\mathcal{R}_1 = \{&\text{“}id_1 : id_{input}\text{”}, \ldots, \text{“}id_k : id_{input}\text{”},\\&\text{“}id_{input} : x_1 : \cdots : x_k : id_g\text{”}\}\end{aligned}$$

This way, a new tuple containing the input vector $(x_1, \ldots, x_k)$ is created and it triggers the execution of g, which generates a tuple with index id_g containing the result.
Step *Hypothesis: for every $f^{(i)}$, with $i \leq n-1$, it exists a set $\mathcal{R}_i$ of specifications to generate $f^{(i)}$.*
We prove that R_n exists for $f^{(n)}$. In the most general case, $f^{(n)}$ can be written as following:

$$\begin{aligned}&f^{(n)}(x_1, \ldots, x_k) =\\&g(f_1(x_1, \ldots, x_k), \ldots, f_j(x_1, \ldots, x_k)) \text{ with } j \geq 1\end{aligned} \tag{7}$$

where f_i either equals to a variable x_h (for any $h \in \{1, \ldots, k\}$) or it is a function $f_i^{(l)}$ with $l \leq n-1$. For the sake of completeness, since a service can use as input any subset of the elements produced by another service, Eq.(7) should be written making use of additional functions to select the input parameters for service functions, in order to make compositions compatible:

$$\begin{aligned}&e_n^{(i_1, \ldots, i_m)}(x_1, \ldots, x_n) = (x_{i_1}, \ldots, x_{i_m})\\&i_m \in \{1, \ldots, n\}\ \forall m \in \{1, \ldots, n\}\end{aligned}$$

As previously mentioned, agent receive notifications for tuples containing at least one input parameter and at a later stage they check the existence of the whole input vector, so functions $e_n^{(i_1, \ldots, i_m)}$ have been implicitly hidden in functions $f^{(i)}$ to make the equation more readable. There are two possible cases.
Case $j > 1$ If f_i equals to x_h then the latter is contained into a tuple with index id_h, we add “$id_h : id_{temp}$” to $\mathcal{R}_n$. In the second case, if the tuple containing the output of $f_i^{(l)}$ has index id_{f_i} (for the inductive hypothesis the index in defined in one specification of $\mathcal{R}_l$), we add to $\mathcal{R}_n$ the specification “$id_{f_i} : x_1 : \cdots : x_n : id_{temp}$”. Moreover, for the inductive hypothesis, for each $f_i^{(l)}$ there exists a set of specification $\mathcal{R}_l$; we add all the elements of $\mathcal{R}_l$ to $\mathcal{R}_n$. Now all the tuples used as input by g have index id_{temp}, we have to merge them within a single tuple, so we finally add “id_{temp}, id_{result}” and “$id_{temp} : y_1 : \cdots : y_j : id_g$” to $\mathcal{R}_n$; the first one combines all the inputs for g into a single tuple, which will trigger the execution of g. The second one assigns the index

id_g to the tuple containing the value of the composition, where $y_i = f_i(x_1, \ldots, x_k)$ for all $i = 1, \ldots, j$ (they can be vectors, so for sake of clarity, in the specification we consider their expansion in basic elements).

Case $j = 1$ In this case we have $f^{(n)}(x_1, \ldots, x_k) = g(f^{(n-1)}(x_1, \ldots, x_k))$ and for the inductive hypothesis the index of the tuple containing the result of $f^{(n-1)}$ must have a index defined in $\mathcal{R}_{n-1}$, let suppose id_1. Then it is sufficient to add the specification "$id_1 : y_1 : id_g$" to $\mathcal{R}_n$ to produce the tuple (with index id_g) containing the result of the composition. Again, y_1 is the type of the value generated by $f^{(n-1)}$. ■

E. Complexities of our approaches.

The complexities of our approaches are simple to calculate if we do not consider metrics depending on the implementation of our model: this is the reason why, at first, we are interested in the number of tuples inserted in the tuple space used to generate an answer, since this metric is strictly related to the number of services automatically executed to to realise the desired composition. Let us assume that each service agent generates a result that can potentially trigger the execution of at most m service agents (information depending on the topology of relations among services represented in the tuple space); if a request generated by an application involves the composition of d services then the complexity of the first approach is:

$$c_{1,tuples} \in O(d^m)$$

By using the second and the third approach we obtain:

$$c_{2,tuples} = c_{3,tuples} \in O(d)$$

It is clear that in these cases, thanks to the discovery process, the request is going to require just the execution of services which take part in the generation of the final result, so their complexity is linear. Regarding the approach for services with more than one input, let assume that h is the maximum number of inputs accepted by a service. By using the activator property, the complexity $c_{n>1,tuples}$ is:

$$c_{n>1,tuples} \in O(hd) = O(d)$$

Since we are dealing with a chemical tuple space, we may also be interested in the number of chemical interactions that are fired in each approach; so we know consider the number of bonds triggered by our algorithms. By using the same variables defined above, we obtain that the complexity $c_{1,bonds}$ of the first approach is:

$$c_{1,bonds} \in O(d^m)$$

because a notification is sent for each service invocation. For the second approach we have to take into account the complexity for the discovery process and for the request process, so we obtain:

$$c_{2,bonds} \in O(d + m)$$

The first term is due to interactions that take place among service agents to spread the "neighbours" properties, whereas the second term is related to the incremental construction of the answer. If we assume that the number of services does not change over time or this term is not relevant, we obtain the same complexity associated with the number of tuples inserted in the tuple spaces. Similarly, for third approach we obtain:

$$c_{3,bonds} \in O(h(d + 1) + m) = O(hd + m)$$

Algorithm 4: Request process - answer: service agent A_i

```
OnServiceRequest():
  S̄_seq = read "activator" value from tuple;
  if (firstElementOf(S̄_seq) not equals to A_i)
  then
  |  wait for another notification;
  end
  x_i = read input value from tuple;
  calculate y_i = f_i(x_i);
  removeFirstElement(S̄_seq);
  S̄'_seq = S̄_seq;
  insert T̄_ri = (P̄_r1, P̄_r2) where _r1 = ("y_i", y_i)
                      P̄_r2 = ("activator", S̄'_seq);
OnServiceDeleteNotification():
  identify agent A_z removed from tuple space;
  remove P̄_zl from L_i;
  calculate H where
  H = {e = "s_1 : s_2 : s_3 : A_j : s_4" in V_il : V_zl ∈ L_i};
  set V_il = (e_1, ..., e_n) where
    e_j in V_zl ∨ e_j in H, V_zl ∈ L_i ∀j = 1, ..., n;
```

V. Implementation

A. SAPERE middleware

For the implementation of our tests, we have used the SAPERE middleware [9], a framework that aims to support the decentralised deployment and execution of distributed adaptive and self-aware applications. The SAPERE model is composed of: an *LSA space*, *Eco-laws*, an *Eco-laws engine* and *agents*. The LSA space is a lightweight tuple space that stores structured tuples named Live Semantic Annotations (LSA), supporting their injection, removal, reading and update. The LSA space is hosted in each node of the network and LSAs can be moved from an LSA space to another. LSAs are manipulated by chemical reactions named Eco-laws; the Eco-laws engine is the active component that takes care of activating Eco-laws periodically or when preconditions for their invocations are satisfied. Beyond Bond, Decay and Aggregate there exists an additional eco-law Spread

Name	W.S. 1	W.S. 2	W.S. 3	W.S. 4	W.S. 5
Input	gps-data	gps-data	city	nation	nation
Output	city	nation	weather	UTC	season

Table I: Summary of our Web services.

for spreading LSAs among several LSA tuple spaces. Each LSA is associated with an agent, a Java external entity which interacts with the tuple space receiving notifications for tuples and inserting, deleting, and updating them. An application (for example a service) that wants to interact with others implements one or more agents, uses them to receives input data and to share the related outputs.

B. Case study

In order to validate our approach, we used the SAPERE middleware to implement a case study with the approaches discussed in this paper. We have developed five Web services summarised in Table I. All Web services accept one input parameter and produce one output and they provide several information about a city, like weather forecast and Coordinated Universal Time. We want to map this system in our model in order to let self-compositions of services produce data suitable for an application providing the GPS position of the device which it is running on. In section IV simple input-output values have been employed but in real applications Web services have to be represented in the tuple space using more descriptive interfaces. One of the most easiest way to do it consists in using a tag-based system to describe them, because of its ease and expressiveness in defining values that will be matched to instantiate a bond. Table II shows how interfaces are represented in LSAs using an XML specification carrying additional information. Depending on the selected approach, the parameters reported

Name	Input/Output
W.S. 1	<param name="gps.data" type="vector.double(2)" /> <param name="city" type="string" />
W.S. 2	<param name="gps.data" type="vector.double(2)" /> <param name="nation" type="string" />
W.S. 3	<param name="city" type="string" /> <param name="weather" type="string" />
W.S. 4	<param name="nation" type="string" /> <param name="UTC" type="time" />
W.S. 5	<param name="nation" type="string" /> <param name="season" type="string" />

Table II: Interfaces of services interacting with the LSA space.

in Table II may be inserted in the tuple space during the invocation of a service or during the discovery and request process. Thanks to the shared tuple space, to be reachable by the others each service has just to instantiate an agent that injects a tuple with the mentioned parameters. By selecting the first approach, the development of applications and service agents is simple. We are interested in using the

Name	Offering/Requesting with "*" operator
application	<param name="gps.data" type="vector.double(2)" /> <param name="weather" type="string" />

Table III: Implementation of an application agent interacting with the LSA space.

Web services reported above to get weather forecasts for a place whose latitude and longitude are known. The code of the application is composed of a few lines of Java: it uses the SAPERE middleware to create an agent that injects a tuple with the values reported in Table III; the second property of that tuple is filled with the "*" operator, in order to receive the notification for a bond with W.S.3 when it produces the weather forecast. When the agent inserts the LSA, the SAPERE middleware automatically realises the composition by using the mechanisms discussed so far; the application agent has just to read the weather forecast from the tuple for which it receives a notification bond. Even the code of a service agent is quite short: during the initialization, each service creates an agent that insert an LSA containing a property requesting the input parameters for the service. When the service agent receives a notification for a bond, it reads the input variables from the tuple, it generates a SOAP request for the Web service and finally it inserts the related result in a new LSA when the latter is available. At the end of the process, the LSA space contains the LSAs reported in Table IV. Tuples with index belonging to range 1-5 are created during the initialization of service agents and they are used to obtain the input parameters for each service. When the application agent requests the weather forecast, it inserts tuple 6, passing a vector filled with latitude and longitude. This property triggers the execution of all services; tuples 7 and 9 take part in the composition of the result requested by the application agent, whereas tuples 8, 10 and 11 are inserted as a "side effect", that is they contain information that will be no useful. When tuple 9 is inserted, a notification bond is delivered to the application agent, which can read the result without dealing with its generation within the LSA space.

When the second approach is selected, the LSA space is quite similar to that one reported in Table IV but tuples contain more complex properties mentioned in section IV-B. Moreover, in this second case, tuples 8, 10 and 11 are not created because the application agent uses the activator property to select the right path to produce the composition that it is looking for. Compared to the second approach, the first one usually invokes unnecessary services, wasting memory resources and computing capacities of platforms hosting services. Nevertheless, it might turn to be the faster when several data are requested; in this case, injecting directly the LSA and waiting for results may be more convenient than resolving several shortest-path problems if the number of services that have to be invoked is approximatively the

Tuple	Properties
Tuple 1	[<param name="gps.data" type="vector.double(2)" />,"*"]
Tuple 2	[<param name="gps.data" type="vector.double(2)" />,"*"]
Tuple 3	[<param name="city" type="string" />,"*"]
Tuple 4	[<param name="nation" type="string" />,"*"]
Tuple 5	[<param name="nation" type="string" />,"*"]
Tuple 6	[<param name="gps.data" type="vector.double(2)" />, (46.17502, 6.14010)] [<param name="city" type="string" />, "*"]
Tuple 7	[<param name="city" type="string" />, Sydney]
Tuple 8	[<param name="nation" type="string" /> Australia]
Tuple 9	[<param name="weather" type="string" />, Cloudy, Max Temp=17 C, Min Temp= 11 C]
Tuple 10	[<param name="season" type="string" />, Spring]
Tuple 11	[<param name="UTC" type="time" />, 17:59 UTC + 9]

Table IV: LSA space at the end of the request for weather forecast.

number of services in the tuple space.

VI. Conclusion and future work

In this paper we defined three approaches to realise self-composition of services by using a chemical tuple space where generic compositions of services are automatically produced as products of several interactions of services, in a distributed way, without any coordinator entity. Moreover, if an application is requesting a result that has been already computed, it will directly access to it, reducing the time to receive the answer. The system reacts automatically when services are added or removed and services involved in compositions are automatically executed in parallel as soon as their input tuples appear in the space. Interactions among tuples are performed using a chemical approach, where the presence of particular elements (properties within tuples) fires a reaction (bond) with other tuples in the space. This means that there must exists a background knowledge about property names in order to let application and services built on top of the space interact reciprocally. In this context, future works about logic specification of service interfaces could pave the way to logic based chemical reactions, where interactions are fired if specific logic statements are satisfied. In our work we have implicitly supposed that the tuple space of the model was distributed and services could be located in several nodes; nowadays efficient implementations of distributed tuple spaces are available for several system architectures ranging from clusters ([11]) to mobile architectures ([12]). Additionally, future works will also focus on specific implementations of the proposed approaches for non-distributed implementations: for example, in the SAPERE middleware each node of the network has its own LSA space and tuples can move across several tuple spaces by using gradient based mechanisms, thus supporting remote queries and answers.

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] J. Rao and X. Su, "A survey of automated web service composition methods," in *Proceedings of the First international conference on Semantic Web Services and Web Process Composition*, ser. SWSWPC'04. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 43–54. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30581-1$_5$

[2] G. Grondin, N. Bouraqadi, and L. Vercouter, "Madcar: An abstract model for dynamic and automatic (re-)assembling of component-based applications," in *Component-Based Software Engineering*, ser. Lecture Notes in Computer Science, I. Gorton, G. Heineman, I. Crnkovi, H. Schmidt, J. Stafford, C. Szyperski, and K. Wallnau, Eds. Springer Berlin Heidelberg, 2006, vol. 4063, pp. 360–367. [Online]. Available: http://dx.doi.org/10.1007/11783565$_2$8

[3] M. Hellenschmidt, "Distributed implementation of a self-organizing appliance middleware," in *Proceedings of the 2005 joint conference on Smart objects and ambient intelligence: innovative context-aware services: usages and technologies*, ser. sOc-EUSAI '05. New York, NY, USA: ACM, 2005, pp. 201–206. [Online]. Available: http://doi.acm.org/10.1145/1107548.1107600

[4] Y. Gabillon, G. Calvary, and H. Fiorino, "Composing interactive systems by planning," in *Proceedings of the 4th French-speaking conference on Mobility and ubiquity computing*, ser. UbiMob '08. New York, NY, USA: ACM, 2007, pp. 37–40. [Online]. Available: http://doi.acm.org/10.1145/1376971.1376979

[5] N. Bonjean, M.-P. Gleizes, C. Maurel, and F. Migeon, "SCoRe: a Self-Organizing Multi-Agent System for Decision Making in Dynamic Software Developement Processes (short paper)," in *International Conference on Agents and Artificial Intelligence (ICAART), Barcelonne, 15/02/2013-18/02/2013*, 2013.

[6] R. Frei, T. erbnu, and G. Marzo Serugendo, "Self-organising assembly systems formally specified in maude," *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–20, 2012. [Online]. Available: http://dx.doi.org/10.1007/s12652-012-0159-2

[7] C. Di Napoli, M. Giordano, Z. Németh, and N. Tonellotto, "Using chemical reactions to model service composition," in *Proceedings of the second international workshop on Self-organizing architectures*, ser. SOAR '10. New York, NY, USA: ACM, 2010, pp. 43–50. [Online]. Available: http://doi.acm.org/10.1145/1809036.1809047

[8] M. Viroli and M. Casadei, "Chemical-inspired self-composition of competing services," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 2029–2036. [Online]. Available: http://doi.acm.org/10.1145/1774088.1774514

[9] G. Castelli, M. Mamei, A. Rosi, and F. Zambonelli, "Pervasive middleware goes social: The sapere approach," in *Proceedings of the 2011 Fifth IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops*, ser. SASOW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 9–14. [Online]. Available: http://dx.doi.org/10.1109/SASOW.2011.6

[10] F. Zambonelli, "Self-aware pervasive service ecosystems," *Procedia Computer Science*, vol. 7, pp. 197 – 199, 2011.

[11] A. Atkinson, "Tupleware: a distributed tuple space for the development and execution of array-based applications in a cluster computing environment," Ph.D. dissertation, University of Tasmania, Australia, 2010. [Online]. Available: http://eprints.utas.edu.au/9996/

[12] E. Sarigöl, O. Riva, and G. Alonso, "A tuple space for social networking on mobile phones," in *ICDE*, 2010, pp. 988–991.