UNIVERSIDAD POLITECNICA DE VALENCIA

PROS
Centro de Investigación en Métodos de Producción de Software

Technical Report

A graph-based definition of a social network for the composition of services by end-users

Pedro Valderas, Victoria Torres and Vicente Pelechano

Ref. #:	**ProS-TR-2019-01**				
Title:	**A graph-based definition of a social network for the composition of services by end-users**				
Author (s):	**Pedro Valderas, Victoria Torres and Vicente Pelechano**				
Corresponding author (s):	**Pedro Valderas, pvalderas@pros.upv.es**				
Document version number:	**2.0**	Final version:	**No**	Pages:	**21**
Release date:	**May 2019**				
Key words:	**Service composition, End-user development, Social Network**				

Universitat Politècnica de Valencia · Camino de Vera s/n · Edificio 1F · 46022 Valencia Spain ·
T. +34 96 387 70 07 Ext. 83530 · M. + 34 619 543 623 · F. +34 96 387 73 59 · info@pros.upv.es · www.pros.upv.es

1. Introduction

Currently, there exist a myriad of end-user environments that face the challenge of involving end-users in the process of service creation (e.g. Daniel et al., 2009; Danado & Paternò, 2014; Aghaee and Pautasso, 2014; Valderas et al., 2017). However, only few of them consider an aspect that end-users demand on current software solutions: social support. Nowadays, millions of people use social networks such as Facebook or Twitter to share with others what is happening in their lives. Messages, images, videos or links are continuously spread through the Internet in order to make people feel that they are connected to others. In the same way, social networks allow people to share their relationships: who are their friends and how many they have, who are they relatives, whether or not they have a sentimental relationship, and so on. Even more, they also share relations with things they have or like. We can find applications that allow people to share books[1], products[2], car journeys[3], homes[4], etc. So, considering this scenario, why not to share also the services compositions with other users through social networks?

Indeed, we think that a social structure created specifically to support end-users in the composition of services can introduce several benefits. First, it can be a valuable mechanism to facilitate end-users to discover services instead of relying on typical internet discovery solutions that cannot scale to the increasingly amount of available (e.g. Al-Masri & Mahmoud 2007, Santanche et al. 2006). For instance, social relationships can be used to allows end-users to browse services within a structure they perfectly know (social networks are currently one of the most used mobile apps (Nielsen 2015)); and they can be also exploited to make service recommendations based on friends' interests. Second, a social network can help end-users to share knowledge with other end-users with the purpose of improving their skills in composing services. Finally, it can also be a collaboration space among end-users and developers, allowing the combination of the innovation and creativity of end-users with the expertise of developers.

In this research report, we propose a graph-based definition of a social structure that characterizes the activity of composing services by end-users. We also define how the underlying connections that are defined in the social structure can be exploited to both (1) help end-users to discover services by browsing through social structure's connections; and (2) recommend services to end-users during the composition activity.

2 Some Foundations about Social Service Composition by End-Users

In this section, we introduce the foundations on service composition by end-users that helped us to characterize the social network that has been proposed in this work (further presented in Section 4).

Hung et al. (2004) defines a web service as an autonomous unit of application logic that provides either some business functionality or information to other applications through an Internet connection. In Service-Oriented Computing (SOC), developers use services as fundamental elements in their application-development processes. Services are platform and network independent operations that clients or other services invoke (Milanovic & Malek, 2004).

Foundation 1. Services are logic units that can be invoked by users or other services.

Currently, the rising of the Internet of Thing (IoT) paradigm has introduced new services that do no operate only in the logic space. IoT services are related to smart devices that provides the functionality of the service (Huo and Wang, 2016). Typically, these services are implemented upon the REST architectural style (Fielding, 2000) by embedding specific servers on smart devices.

[1] http://www.goodreads.com/genres/social-media
[2] http://es.wallapop.com/
[3] https://www.blablacar.co.uk/
4 https://www.homeexchange.com/

Foundation 2. Some services depend on specific devices to provide their functionality

Note that some smart devices are mobile or wearables (e.g. smart phones and watches, glasses, devices incorporated into clothing, and so on) and they can go with the user in order to provide their service wherever the user is (Steinbock, 2005). Other services, however, depend on devices that are physically deployed in a specific space and are highly coupled with the physical environment where they are executed. For instance, smart buildings (Snoonian, 2003) provide services to control lighting, temperature, doors, and so on. Although a user can consume these services through the Internet, their execution depends on the physical space where the smart devices are deployed.

Foundation 3. Some services are highly coupled with a physical location.

According to (Papazoglou et al. 2006) the visionary promise of SOC is a world of cooperating services composed to create flexible dynamic business processes and agile applications that may span organizations and computing platforms. Service composition accelerates rapid application development, service reuse, and complex service consummation (Ding et al., 2008). In this way, developers can solve complex business problems by combining and ordering available basic services to best suit their problem requirement.

Foundation 4. Basic services are composed to create more complex ones in order to properly support business requirements.

There exist several solutions to help developers in the composition of services (Milanovic & Malek 2004). However, the explosion of the number of web services and APIs exposed through the Web has accentuated the need for allowing end-user to create their own service compositions (Yu et al. 2012). As introduced in previous section, we can find a myriad of environments focused on allowing end-user to compose services by their own.

Foundation 5. Service compositions are created by both developers and end-users

In the context of end-user development (Liberman 2006) it is well-known that end-users have many difficulties to create solutions from scratch. It is a good practice to provide predefined elements that can be taken as a basis to define new ones (Segal 2005). The composition of services is not an exception. Environments that support the composition of service by end-users usually provide them with predefined compositions to facilitate the creation of new ones (e.g. Workflow.is, 2018; Zapier, 2018; IFTTT, 2015; Soriano et al., 2008).

Foundation 6. End-users platforms for composing services provide predefined compositions in order to facilitate the creation of new ones.

Independently of the way in which end-users create a composition of services, from scratch or taking as a basis a predefined one, they need to find, select and include the services they need. According to the analysis done in previous section, some approaches (e.g. Zapier, 2018; IFTTT, 2015; Maaradji et al., 2010) introduce mechanisms based on recommendations to help end-users to discover the service they need. In the context of recommendation systems, one of the most used techniques is based on tags or keywords (Milicevic et al., 2010).

Foundation 7. Tag-based recommendation is widely used to help users to find content that fits their preferences.

3 Socializing the Composition of Services

Next, we present an intuitive characterization of the proposed social network to support end-users in the composition of services. The main goal of this network is creating a collaboration space among end-users

and developers in which both can publish their services, consume the service published by others, and compose new service from the published ones.

Considering the foundations presented above, the main characteristics of this social network are:

- Based on Foundations 1, and 4, the social network considers services of two types: *Basic Services*, which are executable logic units implemented by programming activities; and *Composed Services*, which are created through the composition of other services. There exists a relationship among services that indicates that one *Composed Service* includes another service in its definition.
- Based on Foundation 5, it considers two types of users (End-User and Developer). We introduce the possibility of creating a social relationship between two users in order to define the interest of a user in the composition activity of another one.
- Based on Foundation 5, *Basic Services* are created by developers and Composed Services are created by End-users and/or Developers. The social network includes a relationship that indicates that a user is the author of a service,
- Based on Foundation 6, a *Composed Service* can be taken as a basis to create a new *Composed Service*. Thus, there exists a relationship that indicates that one *Composed Service* has been created from another one.
- Based on Foundations 2, 3, and 7, a service can be characterized, among other data, by the devices required for its execution, by a location, or by a set of semantic keywords that describe the execution logics.

Once we have stated the main characteristics of our social network, we have to translate them into a social structure using the proper notions. In Boyd & Ellison (2007), an analysis of current social networks is presented, and authors explain that a social network is characterized by allowing individuals to:
1. Construct a public or semi-public **profile** within a bounded system.
2. Share **content** with a list of connections and perform **actions** with the content shared by others.
3. Articulate a list of user **connections** that can be viewed and traversed, as well as the list of connections made by others.

Although initially defined for social relationships among humans, these three basic characteristics can be adapted in the context of our social network.

3.1 Profiles

We propose a social network where users and services coexist in such a way they can connect to each other. Thus, we need to support profiles of two types:

- **User Profiles.** According to Boyd & Ellison (2007) user profiles usually includes descriptors such as *name*, *birthday*, *location*, *interests*, and a textual description about the user. It is also encouraged the inclusion of a *photo*. In summary, it includes data that can be used to characterize and "know" the user. This data is enough to create a user profile for our purpose if we instantiate it for the goal of composing services. For instance, while we think the birthday date is not required, the user type (Developer or End-user), on the contrary, should be indicated. In addition, the user interests should be related to domains of the services they are interested in.
- **Service Profiles.** With regard to service profiles, we must define analogous data that could help users of the social network (both Developers and End-users) to "understand" services, i.e. to describe services in such a way that users can figure out their internal behaviour. Based on some existing service profiles (Ermagan & Krüger, 2007; Amir & Zeid 2004; Paolucci et al., 2002; Klusch & Sycara, 2001) and the foundations introduced above, we propose a service profile defined from the following data:
    - *name*
    - *type*: Basic or Composed

- *description*: a brief, human-readable description of the service
- *inputs*: values that are required to execute a service
- *output*: value obtained after the execution of a service
- *semantic tags*: list of keywords that characterize the internal behaviour of a service
- *devices' dependencies*: list of physical devices that are needed to execute a service.
- *location*: geographic scope of the service, i.e. a specific space in which the devices the service depends on are physically deployed

3.2 Content and Actions.

Social network users publish content (e.g. multimedia material) and perform actions with the content published by others (e.g. give their opinion, republish it, and so on). Considering the characterization presented above, the content that users can publish in the proposed social network are services, and the actions that other users can do with this content are the following:

- *Include* a service in the composition of a new one;
- *Customize* a Composed Service by taking it as a basis to create a new service.

While the first action implies using a service as a black box, without accessing its internal definition, the second action, as explained above, is targeted only to those services that are created by composing existing ones. This latter action allows users to access and modify the definition of a composition created by other users. Note that this is a valuable mechanism to share knowledge among end-users. As commented above, one of the most important guidelines in End-user Development is providing a catalogue of predefined items that can be taken as a basis (Segal 2005). Thus, we provide end-users with a catalogue of compositions that is made up of all those services composed by other end-users.

3.3 Connections

In a typical social network, connections refer to relationships among humans (users). In addition to this type of relationships, our social network also represent relationships between services, and between users and services.
The connections supported by our social network are the following:

- A *follower* relationship that connect one user to another in order to represent that a user is interested in the composition activity of another user.
- An *author* relationship that connect a user with a service in order to indicate that the user is the author of a service.
- *inclusion* and *definedFrom* relationships that connect two services in order to indicate that a Composed Service includes other services (*inclusion*), and that a Composed Service has been created taking as a basis another Composed Service (*definedFrom*).

4 Social Network Definition

In this section, we present a semi-formal description of a social network to support end-users in the composition of services. Social networks are usually represented as graphs (Wellman, 1988). Consequently, we propose to define our social network as a directed, typed, constrained and attributed graph. Subsection 5.1 presents some notions about graph in order to understand the type of graph that we use. Subsection 5.2 introduces the definition of the social network. Subsection 5.3 analyzes the creation of the explicit and implicit relationships that define the social structure.

4.1 Some basic notions about graphs

This section introduces some intuitive definitions about the notion of graph. In order to access formal foundations of these definitions see (Heckel, 2004; Ehrig, 1979; Löwe, 1993).

A **graph** is an ordered pair *G=(V,E)* where *V* is a set of elements called vertices, nodes, or points, and *E* is a set of elements called edges, arcs or lines, which are 2-element subsets of *V*. Typically, a graph is depicted in diagrammatic form as a set of dots for the vertices, joined by lines or curves for the edges. Furthermore, there exist two functions: a *source* function which indicates the source vertex of an edge and a *target* function which indicates the target node of an edge.

A **directed graph** is a graph in which each edge symbolizes an ordered, non-transitive relationship between two vertices. Thus, *E* is a set of ordered pairs of vertices, called arrows, directed edges, directed arcs, or directed lines.

A **typed graph** is a graph whose nodes and/or edges are classified by attaching types to them, i.e. there is a function that indicate the type of each graph element. The same type may be assigned to different elements. A typed graph is defined from the sets *V* and *E*, and moreover from: (1) a set T_V of types for vertices and a function $type_V$ which indicates the type associated to a vertex; and/or (2) a set T_e of types for edges and a function $type_e$ which indicates the type associated to an edge.

A **constrained graph** presents constraining functions that are attached to vertices and edges. These functions associate an arbitrary number of constraints to any vertex or edge. Constraints are expression to define cardinality, restrictions on the domain or the co-domain of certain functions, etc. Constrains are usually expressed with first order logic expressions (Orejas, 2008).

An **attributed graph** is a graph in which we attach attributes to its vertices or edges. Each vertex and edge can have attached more than one attribute. Thus, a vertex attributed graph is defined from the sets *V* and *E*, and moreover from: (1) a set V_D of data vertices, which represent the values that can be assigned to attributes; (2) a set E_{NA} of vertex-attribute edges, which contains edges whose source is a vertex of V and whose target is a data vertex of V_D. Finally, an additional aspect that must be considered in order to correctly define an attributed graph is that we must associate to the graph an algebra over a data signature DSIG, in the sense of algebraic signatures (Ehrig & Mahr, 1985). In this signature, we distinguish a set of attribute value sorts. These sorts are used to define the valid values that can be assigned to attributes (elements of V_D). In order to see formal definitions about this aspect see (De Lara et al., 2005).

4.2 Semi-formal definition of the proposed social network

To define the proposed social network, we use a directed, typed, constrained and attributed graph where just attributes for vertices are considered in order to simplify its definition. First, we present the elements of the social network (i.e. users and services) and the connections among them. Next, we define the data included in the profiles of these elements.

Elements and connections:

- A set of vertices *V* and a set of directed edges *E*. The functions *source* and *target* which indicate the source and target vertex of an edge.
- We propose two types of vertices in order to represent users and services (profiles). Thus, there exists a set of types for vertices T_V*={User, Service}* and a function $type_V$ which indicate the type associated to a vertex.
- We also propose two types of users (End-users and Developers) and two types of services (basic and composed). Thus, there exists: a set of user subtypes T_u*={End-User, Developer}* and a function $subtype_u$ which indicates the subtype associated to a vertex whose type is *User*; a set of service subtypes T_s*={Basic, Composed}* and a function $subtype_s$ which indicates the subtype associated to a vertex whose type is *Service*.
- There are different types of edges in order to represent the relationships among users and services. Thus, there exists a set of types for edges T_e*={follower, author, includes, definedFrom}*; and a function $type_e$ which indicate the type associated to an edge. These types of edges are used to

represent relationships between two users, between two services, and between a user and a service. We need some constraints that properly define their use:

- *follower*: These edges represent the interest of a user A on the service composition activity done by a user B. Thus, they can only be defined between two users:

  $\forall e \in E \mid type_e(e)=follower \rightarrow type_v(source(e))= User \,\&\, type_v(target(e)) = User$

- *author*: These edges represent the authorship of a service by a user. Thus, they must connect a user with a service:

  $\forall e \in E \mid type_e(e)= author \rightarrow type_v(source(e)) = User \,\&\, type_v(target(e))= Service$

  In addition, note that Basic Services can only be created by developers. Thus:

  $\forall e \in E \mid type_e(e)=author \,\&\, subtype_s(target(e)) = Basic \rightarrow subtype_u(source(e))= Developer$

- *includes*: These edges represent that a Composed Service includes another service in its definition. The source of these edges must be a Composed Service and the target another service:

  $\forall e \in E \mid type_e(e)=includes \rightarrow type_v(source(e)) = Service \,\&\, subtype_s(sourcet(e))= Composed \,\&\, type_v(target(e))= Service$

- *definedFrom*: These edges represent that a Composed Service has been defined by taking the definition of another Composed Service as a basis. They can only be defined between Composed Services:

  $\forall e \in E \mid type_e(e)=definedFrom \rightarrow type_v(source(e))= Service \,\&\, subtype_s(sourcet(e))= Composed \,\&\, type_v(target(e))= Service \,\&\, subtype_s(target(e))= Composed$

– To represent the order in which services are included in a Composed Service we complement each Composed Service with an ordered set of *includes* edges, and the relation "<" defines the order among elements according to the edge creation time. Thus, $e_1 < e_2$ specifies that e_1 was created before e_2. The function *definition* returns the ordered set associated to each Composed Service. Thus, the definition of a Composed Service cs_i is defined as:

$$definition(cs_i)= \{e_1 < e_2 < e_3 < \ldots < e_N\}$$

Functions *first* and *last* indicate the first and last *includes* edges of the ordered set associated to a Composition Service. Functions *previous* and *next* indicate the previous and next *includes* edges of another one included into the ordered set associated to a Composed Service. An *includes* edge can only be included in one and only one ordered set:

$$\forall e \in E \mid type_e(e) = includes \rightarrow e \in definition(source(e)) \,\&\, \nexists s \in V \mid type_v(s) = Service \,\&\, subtype_S(s) = Composed \,\&\, s != source(e) \,\&\, e \in definition(s)$$

Profile Data:

– There is a set of data vertices V_D and a set of vertex-attribute edges E_{VA} together with the $source_{va}$ and $target_{va}$ functions that indicate the source and target vertex of each vertex-attribute edges.

– We include a set of types for vertex-attribute edges T_{vae}=*{name, location, interest, description, input, ouput, semantics, device}* and a function $type_{vae}$ which indicates the type associated to a vertex-attribute edge. Thus, the attributes that represent the types associated to each vertex-attribute edge are:

  - *name:* it associates a name to a user or a service:
    $\forall e_{ae} \in E_{VA} \mid type_{uae}(e_{ae})=name \rightarrow source(e_{ae}) \in V \,\&\, target(e_{ae}) \in V_D$

  - *location*: represent the location of a user or the physical scope of a service:
    $\forall e_{ae} \in E_{VA} \mid type_{vae}(e_{ae})=location \rightarrow source(e_{ae}) \in V \,\&\, target(e_{ae}) \in V_D$

- *interest*: it represents a domain in which a user is interested in. This vertex-attribute edge can only be associated to users:
  $\forall e_{ae} \in E_{VA} \mid type_{vae}(e_{ae})=interest \rightarrow type_v(source(e_{ae}))= User \,\&\, target(e_{ae}) \in V_D$
- *description*: it associates a textual description to a user (about me) or a service:
  $\forall e_{ae} \in E_{VA} \mid type_{vae}(e_{ae})=description \rightarrow source(e_{ae}) \in V \,\&\, target(e_{ae}) \in V_D$
- *input*: it associates an input to a service. This vertex-attribute edge can only be associated to services:
  $\forall e_{ae} \in E_{VA} \mid type_{vae}(e_{ae})=input \rightarrow type_v(source(e_{ae}))= Service \,\&\, target(e_{ae}) \in V_D$
- *output*: it associates an ouput to a service. This vertex-attribute edge can only be associated to services:
  $\forall e_{ae} \in E_{VA} \mid type_{vae}(e_{ae})=output \rightarrow type_v(source(e_{ae}))= Service \,\&\, target(e_{ae}) \in V_D$
- *semantics*: it associates a keyword to a service in order to characterizes their semantics. This vertex-attribute edge can only be associated to services:
  $\forall e_{ae} \in E_{VA} \mid type_{vae}(e_{ae})=semantics \rightarrow type_v(source(e_{ae}))= Service \,\&\, target(e_{ae}) \in V_D$
- *device*: it associates a device to a service. This vertex-attribute edge can only be associated to services:
  $\forall e_{ae} \in E_{VA} \mid type_{vae}(e_{ae})=device \rightarrow type_v(source(e_{ae}))= Service \,\&\, target(e_{ae}) \in V_D$

Example:

Figure 1 shows a graphical example of a partial view of a graph that represents the proposed social network. It is defined by the following sets:

- *V={u1, u2, u3, s1, s2, s3, s4, s5, s6}*
  $type_v$*(u1-3)=User;* $type_v$*(s1-6)=Service*
  $subtype_u$*(u1 and u2)=End-User;* $subtype_u$*(u3)=Developer;*
  $subtypes_s$*(s1, s4-6)=Composed;* $subtypes_s$*(s2, s3)=Basic;*
- *E={e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11}*
  $type_e$*(e1)=follower;* $type_e$*(e2, e3, e4, e5, e6, e7)=author;*
  $type_e$ *(e8, e9, e10)=includes;* $type_e$*(e11)=definedFrom*
- V_D*={Rome, Valencia, work, office, travel, SmartPhone, John, AtTheUni, emailAccount}*
- E_{VA}*={ e12, e13, e14, e15, e16, e17, e18, e19, e20, e21, e22, e23, e24}*
  $type_{vae}$*(e12, e13, e14, e15)=semantics ;* $type_{vae}$ *(e16, e17, e18)=location;*
  $type_{vae}$*(e19)=input;* $type_{vae}$*(e20, e21)=device;* $type_{vae}$*(e22, e23)=name;* $type_{vae}$*(e24)=interest;*

To simplify the example, most of the vertex-attribute edges required to define profiles completely are omitted. We have also omitted the sources and target functions. The algebra over a data signature to distinguish attribute value sorts is not presented, and all values are considered as String. In a more precise definition, some attribute values would need a different sort. For instance, the location attribute could be specified as a tuple <latitude, longitude> or the input and output attributes could be specified as a tuple <name, type>. Note also that the ordered sets that define Composed Services are not presented. As representative example, we can consider definition(s3)={e9 < e10} indicating that first, e9 was included in s3, and next, e10 was included in s3.

In order to better visualize the graph, we have depicted vertices of type User as black circles, vertices of type Service as black squares, and data vertices as grey ellipsis. Subtypes of users and services are indicated with the initial letter(EU and D for End-user and Developer; B and C for Basic and Composed). In the same was, edges from E are depicted by solid arrows while vertex-attribute edges from E_{VA} are depicted by dashed arrows.

According to Figure 1:

- there are three users in the social network: u1, u2 and u3;
- u1 and u2 are end-users; u3 is a developer;

- the name of u1 is 'John';
- u2 is interested in 'travel'
- u1 is a follower of u2;
- there are six services in the social network: s1, s2, s3, s4, s5, and s6;
- s2 and s5 are basic services; s1, s3, s4 and s6 are composed services;
- u1 has created the service s4;
- u2 has created the services s1and s6;
- u3 has created the services s2, s3, and s5;
- s1 includes the execution of s2 into its definition;
- s3 includes the execution of s2 and s6 into its definition;
- s4 has been defined taking as a basis the definition of s1;
- s4 has been named as 'AtTheUni';
- s5 needs an 'emailAccount' as input;
- the execution of s1 and s5 depends on the location 'Valencia';
- the execution of s3 depends on the location 'Rome';
- the semantics of s4 is related to 'work';
- the semantics of s5 is related to 'work' and 'office';
- the semantics of s1 is related to 'travel';
- the execution of s3 and s6 depends on a 'SmartPhone' device

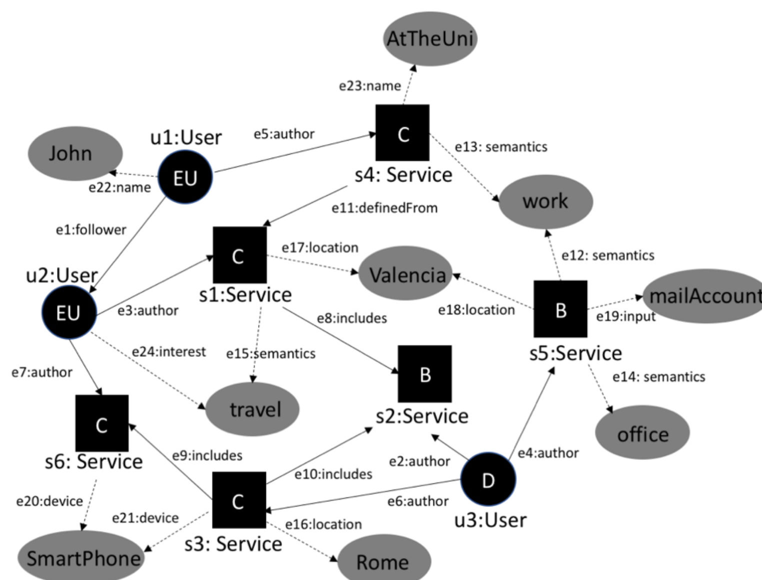

Figure 1. Example of graph representing a social network for service composition by end-users

4.3 Social Structure Creation. Implicit vs Explicit relationships

The above introduce graph represents a social structure to support end-users in the compositions of service. This structure reflects the activity done by users when composing services and ends-up with a graph linking the users according to their interests in composition activities. However, note that the social relations that it supports are created in two different ways:

- Explicit relations: these relations are those created by users explicitly. This is similar to what currently happens in social networks like Facebook when one user accepts the invitation of another one to be friends. In this case, we have:
  - The relationship represented by the *follower* edge between two users. One user must invite another to be its follower and vice versa.

  - The relationships represented by all the vertex-attribute edges. Attributes of a user profile must be explicitly defined by the own user; attributes of a service profile must be explicitly defined by the author of the service.
- Implicit relations: these relations are implicitly inferred from to the activities of the different users. In this case, we have:
  - The relationship represented by the *author* edge between a user and a service. When a user composes a new service, an *author* edge between the user and the service is implicitly created.
  - The relationship represented by the *includes* edge between two services. When a user includes a service when composing a new one, an *includes* edge between the new service and included one is implicitly created.
  - The relationship represented by the *definedFrom* edge between two services. When a user composes a new service by taking as basis an existing one, a *definedFrom* edge between the new service and the existing one is implicitly created.

5 Exploiting social network's relationships to browse and discover services

The main goal of the above-introduced social network is supporting end-users in the composition of services. One of the most interesting features that it provides is the possibility of exploiting the set of relationships that are created to improve the problem of service discovery. The proposed social structure can be used to allow end-users to discover services by browsing profiles, but also to recommend end-users with services during the composition process.

5.1 Browsing services through the social structure

Efficiently supporting end-users in exploring services composed by other end-users is a key requirement of the proposed social network. Exploring services is the process of navigating through available services and acquiring important knowledge of them (Yu et al. 2012) and it heavily relies on how the characteristics of services are represented, organized, and rendered.

With the proposed social network, we provide end-users with a tool that allows them to find services by browsing a set of users and services profiles, which are notions that are familiar for them. Social networks are currently one of the most used mobile apps (Nielsen 2015) and the use of a social structure is familiar to end-users, making the task of finding services easier than current service repositories.

In this sense, a user u_i of the proposed social network can browse:

- The list of followed users. To describe this, we propose de function Followed Users (FU):

  $FU(u_i)=\{ u_j \in V \mid type_v(u_j)=User \ \& \ \exists e \in E \mid type_e(e_j)=follower \ \& \ source(e)=u_i \ \& \ target(e)=u_j \}$

If the user u_i, access the profile of a followed user u_j, it can browse:

- The services created by the followed user u_j. To describe this, we propose the function Followed Created Services (FCS):

  $FCS(u_i, u_j)=\{ s \in V \mid type_v(s)=Service \ \& \ \exists e \in E \mid type_e(e)=author \ \& \ source(e)=u_j \ \& \ target(e)=s \ \& \ u_j \in FU(u_i)\}$

- The services used by the followed user u_j to create new Composed Services. To describe this, we propose the function Followed Included Services (FIS):

  $FIS(u_i, u_j) = \{ s \in V \mid type_v(s)=Service \ \& \ \exists e \in E \mid type_e(e)=includes \ \& \ source(e) \in FCS(u_i, u_j) \ \& \ target(e)=s\}$

If the user u_i accesses the profile of the service s_i, it can browse:

- The services created by the users followed by u_i that are associated to the same location. To describe this, we propose the function Same Location Services (SLS):

  $SLS\ (u_i, s_i) = \{ s_j \in V \mid type_v(s_j)=Service\ \&\ \exists\ e_i, e_j \in E \mid type_e(e_i)=location\ \&\ type_e(e_j)=location\ \&\ source(e_i)=s_i\ \&\ source(e_j)=s_j\ \&\ target(e_i)=target(e_j)\ \&\ \exists\ e_k \in E \mid type_e(e_k)=author\ \&\ target(e_k)= s_j\ \&\ source\ (e_k) \in FU(u_i)\}$

- The services created by the users followed by u_i that share some semantic keyword. To describe this, we propose the function Similar Semantics Services (SSS):

  $SSS\ (u_i, s_i) = \{ s_j \in V \mid type_v(s_j)=Service\ \&\ \exists\ e_i, e_j \in E \mid type_e(e_i)=semantics\ \&\ type_e(e_j)=semantics\ \&\ source(e_i)=s_i\ \&\ source(e_j)=s_j\ \&\ target(e_i)=target(e_j)\ \&\ \exists\ e_k \in E \mid type_e(e_k)=author\ \&\ target(e_k)= s_j\ \&\ source\ (e_k) \in FU(u_i) \}$

- The services created by the users followed by u_i that depend on the same device. To describe this, we propose the function Same Device Services (SDS):

  $SDS\ (u_i, s_i) = \{ s_j \in V \mid type_v(s_j)=Service\ \&\ \exists\ e_i, e_j \in E \mid type_e(e_i)=device\ \&\ type_e(e_j)=device\ \&\ source(e_i)=s_i\ \&\ source(e_j)=s_j\ \&\ target(e_i)=target(e_j)\ \&\ \exists\ e_k \in E \mid type_e(e_k)=author\ \&\ target(e_k)= s_j\ \&\ source\ (e_k) \in FU(u_i) \}$

- The services that the users followed by u_i have defined and include s_i in their definition. To describe this, we propose the function Supporting Services (SS):
  $SS\ (u_i, s_i) = \{ s_j \in V \mid type_v(s_j)=Service\ \&\ \exists\ e_j \in E \mid \&\ type_e(e_j)=includes\ \&\ source(e_j)=s_j\ \&\ target(e_j)=s_i\ \&\ \exists\ e_k \in E \mid type_e(e_k)=author\ \&\ target(e_k)= s_j\ \&\ source\ (e_k) \in FU(u_i)\}$

- The services that the users followed by u_i have included together with s_i in the definition of a same Composed Service. To describe this, we propose the function Co-Work Services (CWS):
  $CWS\ (u_i, s_i) = \{ s_j \in V \mid type_v(s_j)=Service\ \&\ \exists\ e_i, e_j \in E \mid type_e(e_i)=includes\ \&\ type_e(e_j)=includes\ \&\ target(e_i)=s_i\ \&\ target(e_j)=s_j\ \&\ source(e_i)=source(e_j)\ \&\ \exists\ e_k \in E \mid type_e(e_k)=author\ \&\ target(e_k)= s_j\ \&\ source\ (e_k) \in FU(u_i)\}$

- The services that the users followed by u_i have defined taking as a basis s_i. To describe this, we propose the function Same Parent Services (SPS):
  $SPS\ (u_i, s_i) = \{ s_j \in V \mid type_v(s_j)=Service\ \&\ \exists\ e_j \in E \mid type_e(e_j)=definedFrom\ \&\ source(e_j)=s_j\ \&\ target(e_j)=s_i\ \&\ \exists\ e_k \in E \mid type_e(e_k)=author\ \&\ target(e_k)= s_j\ \&\ source\ (e_k) \in FU(u_i)\}$

5.2 Recommending services in composition activities

As commented above, end-user environments usually provide the possibility of composing services in two ways: (1) from scratch and (2) from a predefined composition. We focus on analyzing how social network's relationships can be used to recommend services to end-users in these two situations:

(1) End-users are creating a Composed Service from scratch. In this case, we want to provide end-users with a list of services that can fit their needs before starting the composition of a new service.

(2) End-users want to compose a service from an existing Composed Service as a basis. In this case, we want to recommend the Composed Services that better fit end-users' interests.

In addition to these two situations, service recommendations can also be provided at any time during the composition process, independently of the way end-users create a composition:

(3) End-users have a Composed Service partially defined. In this case, recommendations are focused on providing a list of the most suitable services to be included after the last service included in the Composed Service.

In order to make recommendations in these three situations we propose the following functions:

- Inclusion Number (IN): This function returns de number of times that a user u_i has included a service s_i when composing another service.

  $IN(u_i,s_i)=|\{e_i \in E \mid type_e(e_i)=inclusion \;\&\; target(e_i)=s_i \;\&\; \exists\, e_j \in E \mid type_e(e_j)=author \;\&\; target(e_j)=source(e_i) \;\&\; source(e_j)=u_i\}|$

- Service Interest Level (SIL). This function returns the level of interest that a user u_i has in a service s_i. This level is calculated from the frequency that a service s_i is included in the compositions done by a user u_i.

  $SIL(u_i, s_i)=\frac{IN(u_i s_i)}{\sum_{k=1}^{N} IN(u_i, s_k)}$

- Composed Service Interest Level (CSIL). This function returns the level of interest that a user u_i has in a Composed Service s_i in terms of the interest that the services included in the Composed Service has for the user.

  $CSIL(u_i, s_i)=\sum_{k=1}^{N} SIL(u_i, s_k)\ \forall\, s_k \mid \exists\, e \in E \;\&\; type_e = includes \;\&\; source(e) = s_i \;\&\; target(e) = s_k$

- Service User Similarity (SUS). This function returns the level of similarity that a user u_i has with a user u_j in relation with the interest in service s_i.

  $SUS(u_i, u_j, s_i)=\frac{SIL(ui,si)}{SIL(uj,si)}$

- Global User Similarity (GUS). This function returns the level of similarity that a user a u_i has with a user u_j in a global way, i.e. considering their interest of both in every service.

  $GUS(u_i, u_j)=\sum_{k}^{N} SUS(u_i, u_j, s_k)$

End-users are creating a Composed Service from scratch. . From a single end-user perspective, services can be recommended based on their historical usage, i.e., the most used services are first offered to the user. From a social perspective, we can also recommend the first services used by their followed users in the Composed Services. This last recommendation is defined in Algorithm 1: given a specific user u_i, we obtain all the Composed Services created by its followed users. For each of this Composed Services, we access the service included in first place, and calculate for it a Recommendation Level (RL). RL is calculated from the product between the interest of the followed user in the service (Service Interest Level) and the Global User Similarity between both users. Then, the list of services ordered by this level is returned

Algorithm 1. Recommendation of first services

```
input ui ∈ V & typev(ui)=User
  for each uk ∈ FU(ui)
    for each csk ∈ {s ∈ FCS(ui, uk) & subtypes(s)= Composed}
      sk=target(first(csk))
      RLk = SIL(uk, sk)*GUS(ui,uk)
      Add (sk,RLk) in RecList
    end for
  end for
  Sort RecList in descending order of RLk
output RecList = {(sk, RLk)}
```

End-users want to compose a service by taking an existing Composed Service as basis. From a single end-user perspective, Composed Services can be recommended based on their historical usage, i.e., the Composed Services previously used to compose new one are first offered to the user. From a social perspective, we can also recommend her/him the Composed services built by followed users with similar

interests. This recommendation is defined in Algorithm 2: given a specific user u_i, we obtain all the Composed Services created by its followed users. For each of these Composed Services, a Recommendation Level (RL) is calculated from the product between the Composed Service Interest Level of the followed user and the Global User Similarity between both users. Then, the list of Composed Services ordered by this level is returned.

Algorithm 2. Recommendation of Composed Services

```
input ui ∈ V & typev(ui)=User
  for each uk ∈ FU(ui)
   for each csk ∈ {s ∈ FCS(ui, uk) & subtypes(s)= Composed}
    RLk = CSIL(uk, csk)*GUS(ui,uk)
    Add (sk,RLk) in RecList
   end for
  end for
 Sort RecList in descending order of RLk
output RecList = {(sk, RLk)}
```

End-users have a Composed Service partially defined. From a single end-user perspective, services can be recommended by analyzing the services previously used by end-users to build Composed Services. From a social perspective, we can also analyze the services included in the Composed Services created by their followed users. This recommendation is defined in Algorithm 3. Given a specific user u_i and a service s_i that is the last service included in the composition that u_i is currently creating: first, we access all the users followed by u_i; Next, we obtain all the *includes* edges that target s_i and whose source is a Composed Service created by each followed user; for each edge, we obtain the service associated to the next *includes* edge within the service definition they belong; then, a Recommendation Level (RL) is calculated from the product between the interest of the followed user in the service (Service Interest Level) and the Global User Similarity between both users. Finally, the list of Composed Services ordered by this level is returned.

Algorithm 3. Recommendation of next services

```
input ui ∈ V & typev(ui)=User, si ∈ V & typev(si)=Service
  for each uk ∈ FU(ui)
   for each ek ∈ { eh ∈ E | typee(eh)=includes & target(eh)=si & ∃ ej ∈ E | typee(ej)=author &
target(ej)=source(eh) & source(ej)=uk }
    next=target(next(ek))
    if (next != null) then
      RLk = SIL(uk, next)*GUS(ui, uk)
      Add (next, RLk) in RecList
    end if
   end for
  end for
 Sort RecList in descending order of RLk
output RecList = {(next, RLk)}
```

6. A social infrastructure

In this section, we present a software infrastructure to support the social network presented above. Works such as (Atzori et al 2007), (Chang et al. 2007) or (Guinard et al. 2010) propose architectures to support some kind of social network. They are mainly based on the use of repositories to manage the social data, and some type of access controller to provide client applications with access to this data. Inspired by these works, the software architecture proposed to develop Social-EUCalipTool is composed by the following elements: (see Figure 2):

1. *User* and *Service Registries*: While the *User Registry* is in charge of maintaining the data from the user profiles and the *follower* connections, the *Service Registry* is in charge of maintaining the data

associated to service profiles as well as the connections created between two services (*includes* and *defiendFrom*).

2. The *Social Controller* is in charge of accessing the user and service registries in order to manage all social data. It is composed of two main modules:
   a. *Social Structure Builder*: This component is in charge of accessing the user and service registries and constructing the social structure required by the composition environment. It implements the browsing function presented in Section 5.1.
   b. *Service Recommender*: This component is in charge of analyzing the social structure provided by the *Social Structure Builder* in order to make service recommendations. It implements the functions and recommendation algorithms presented in Section 5.2.
3. *Social Service Composition Environment*: This element constitutes the client application that users interact with to compose and browse services. It is composed of two main modules:
   a. *Service Authoring Tool*: This module provides an editor to allow end-users to compose services. It also allows developers to create basic services.
   b. *Service Browser*: This module provides users with a user interface to browse the social structure provided by the *Social Structure Builder*. Thus, it is in charge of implementing user and service profiles and the connections among them.

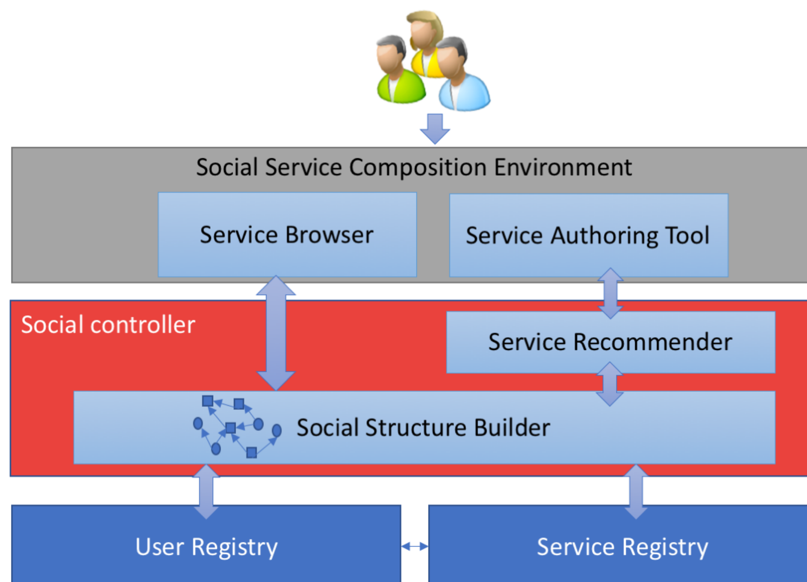

Figure 2. Software architecture to create, execute and *share* services with

7. Evaluation of the Social Controller

In this section, we evaluate the implementation done of the two elements that made up de Social Controller, i.e. the *Social Structure Builder* and the *Service Recommender*. In particular, we want to analyse the completeness and correctness of the functions presented in Section 5, which are implemented by both elements, and the performance of the algorithms implemented by the *Service Recommender*.

Both elements have been implemented as web Java modules deployed on an Apache Tomcat server. The *User* and *Service* registries are implemented with to databases deployed into a MySQL server. Both servers are installed in a MacBook Pro (2,9 GHz Intel Core i5, 8GB RAM and SD disk). The Java modules that implement the *Social Structure Builder* access the MySQL server in order to manage the social structure of

the network. To obtain a balance between memory load and database access, each time a user log into the social network the *Social Structure Builder* accesses the database to create an initialization of the social structure in which the SIL of each service regarding the current user is calculated as well as the GUS of the current user with their followers. Afterwards, a lazy loading design pattern is implemented in order to load the rest of data (e.g. the services included in a Composed Service) when it is needed.

7.1 Correctness and Completeness

In order to evaluate the correctness of the functions implemented by the two modules of the Social Controller, we need to check they return those services and connections and only those, that correspond with the semantic of each function. In the same way, the completeness of each function is evaluated by checking the all the services and connections that are stored in the Social Registry and match with the semantic of each function are returned.

In order to perform all these evaluations, we used JUnit tests. We developed a set of JUnit tests that allows us to evaluate the proper behavior of each function. For instance, the next code shows JUnit test that evaluates the completeness of the SPS (Same Parent Services) function, which is implemented by *Social Structure Builder* and must return the services of the users followed by another one that are all created taking as basis a service of this user. This test is implemented by a method that receives as argument a user id, a service id, and the expected services that should be returned by the SLS functions. This method uses the methods `getUserById` and `getServiceById` that has been previously evaluated. From the service obtained with these two methods we execute the SLS function and use JUnit to compare the obtained list of services with the expected one.

```
@Test
public void SPS_Test(String userID,
                     String serviceID,
                     List<Service> expectedServices){

      SocialStructureBuilder ssb = new SocialStructureBuilder();

      User u=ssb.getUserById(userID);
      Service s=u.getServiceById(serviceID);

      List<Service> sameParentServices=ssb.SPS(u,s);

      assertEquals(expectedServices, sameParentServices);
}
```

In the same way, the following code shows the test that evaluate the correctness of IN (Inclusion Number), which is implemented by the *Service Recommender* and must return the number of times that a user has included a service in the creation of another.

```
@Test
public void IN_Test(String userID,
                    String serviceID,
                    Integer expectedTimes){

      ServiceRecommender sr = new ServiceRecommender();
      SocialStructureBuilder ssb = new SocialStructureBuilder();

      User u=ssb.getUserById(userID);
      Service s=u.getServiceById(serviceID);

      Integer times=sr.IN(u,s);

      assertEquals(expectedTimes, times);
}
```

The completeness and correctness of all the functions implemented by the *Service Recommender* and the *Social Structure Builder* have been evaluated by testing methods like the two one presented above. To do so, we created manually a social structure in the database that implements the *Social Registry*. Next, for each function we manually define the expected results for several cases and use the testing method to check the proper behaviour of the functions.

7.2 Performance of recommendations

In this caser, we want to check if the execution time of the three proposed algorithms and the functions they use is quick enough to ensure a good user experience when using an authoring environment. As a reference, we consider the study done by Nielsen (2010). According to this study, a response time of 0.1 seconds gives the feeling of instantaneous response, which is essential to support the feeling of direct manipulation and increase user engagement and control; a response time of 1.0 second is about the limit for the user's flow of thought to stay uninterrupted.

Due to the lack of benchmarks, the data sets were generated randomly. In order to create an initial data set, we considered some social network statistics: (1) According to Dogtiev (2018), most of the teenagers had only up to 100 followers in Instagram; (2) Some of the data presented by Smith (2018) indicate that users of IFTTT create an average of 6 Applets per user; (3) Analyzing the Programmable Web[5] site, we concluded that there is a Mashup for each 2.5 APIs; (4) Maaradji et al. 2011 did an experiment with mashups with an initial benchmark of 1500 users and mashups made up of 4.5 services in average.

Thus, considering the above statistics, we created an initial data set with 1500 users. Each user had 100 followed users and was the author of 6 Composed Services. Thus, we have a total of 9000 Composed Services (1500 users * 6), and we created 22.500 Basic Services (there was a mashup per 2.5 APIs, 9000*2.5= 22.500). Each Composed Service included 5 services.

The proposed algorithms depend on: (1) the number of followed users; (2) the number of Composed Services created by them; and (3) the number of services included in each Composed Service. Thus, we have done several experiments in which we modify these three parameters and measure the execution time. In each experiment, we have executed the algorithms 1 and 2 for each user, and the algorithm 3 for each user with a randomly generated service. From these executions we calculated the longest execution time for each

[5] https://www.programmableweb.com/

algorithm. We ran each experiment 5 times and calculated the average among the obtained longest execution time obtained each time.

Experiment 1: **different number of followed users**. Taking as base the initial data set presented above we did an experiment in which we varied the number of followed users (100, 300, 500, 700, 1000) and maintained fixed the other two parameters. The initialization time and the average in the longest execution time for each algorithm are shown in Table 1. As we can see, the initialization time is little higher than 0.1s but pretty shorter than 1 second independently of the number of followed users. This is a good time to provide a good user experience in the initial loading of the authoring environment. The execution time of the three algorithms are quite shorter than 0.1 seconds. Algorithm 1 has the shortest response time since we only need to access the SIL and GUS values of a user, which are pre-calculated in the initialization; Algorithm 2 has similar execution time although a little higher than algorithm 1 because we need to calculate the CSIL of each Composed Service created by followed users which implies the addition of the SIL for each included service; the Algorithm 3 has the longest execution time because we need to find the services that are previous to a another one in each Composed Service created by the followed users. This data is not loaded in the initialization so we have to access the database several times, which produces a higher execution time. Figure 3 shows graphically the execution time of algorithms. As we can see, the execution time growth in almost a linear way with the number of followed users. If we considering the execution time of each algorithm obtained for the highest number of followed users (1000), they are all lower than 0.1 seconds which are good values to provide a good user experience.

Table 1. Initialization time and execution time of each algorithm in experiment 1

# followed users	Initialization	Algorithm 1	Algorithm 2	Algorithm 3
100	150.16 ms	2.28 ms	2.72 ms	3 ms
300	150.52 ms	6.36 ms	6.6 ms	8.52 ms
500	151.56 ms	13,4 ms	13.8 ms	15.28 ms
700	152.4 ms	19.56 ms	20.12 ms	22.45 ms
1000	153.5 ms	28.7 ms	29.36 ms	33,2 ms

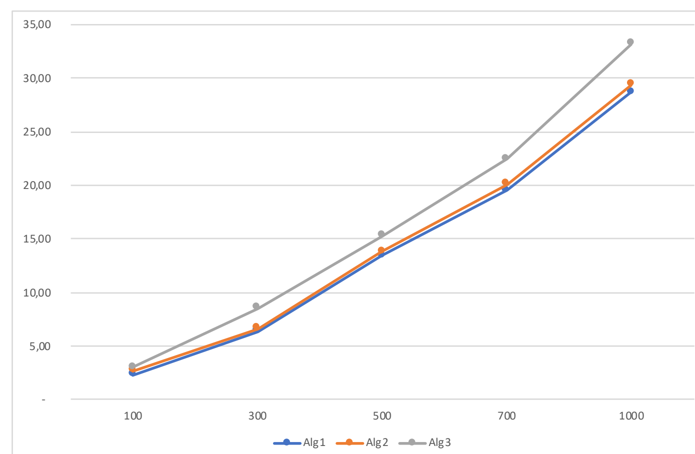

Figure 3. Execution time of each algorithm (in milliseconds, y-axis) for different number of followed users (x-axis)

Experiment 2: **Different number of Composer Service per User**. Taking as base the initial data set presented above we did an experiment in which we varied the number of Composed Services created by each user (6, 10, 15, 20, 25) and maintained fixed the other two parameters. Note that this parameter has a direct effect on the number of services. As this parameter grows, the number of Composed Services and Basic Services increase. The results of this experiment are shown in Table 2. We can see how this parameter has a significant impact over the initialization time. Nevertheless, the initialization time obtained in the

experiment is acceptable to provide a good user experience in the initial loading of the authoring environment. Regarding the results obtained for the three algorithms, the execution time is quite short for the tested values, which lead us to think that a good response time can be obtained when interacting with the authoring environment. Figure 4 shows how the increase in the execution time is less accentuated in this experiment than in the previous one.

Table 2. Initialization time and execution time of each algorithm in experiment 2

Comp Serv/User	Initialization	Algorithm 1	Algorithm 2	Algorithm 3
6	150.16	2.28	2.72	3
10	274	3.43	3.61	3.13
15	520.9	3.9	4.09	3.63
20	628.64	4.55	4.66	4.23
25	751.55	5.6	5.92	4.71

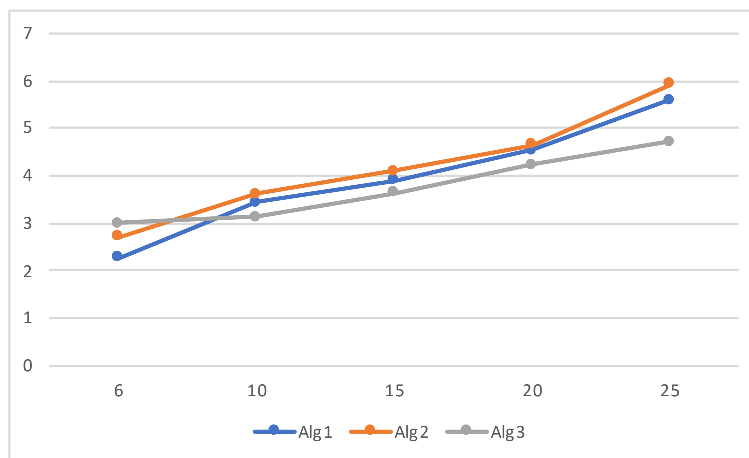

Figure 4. Execution time of each algorithm (in milliseconds, y-axis) for different amount of Composed Services per user (x-axis)

Experiment 3: **Composer Services made up of different number of services**. Taking as base the initial data set presented above we did an experiment in which we varied the number of services included in each Composed Service (5, 10, 15, 20, 25) and maintained fixed the other two parameters. The results of this experiment are shown in Table 3. As happens in the previous experiment, this parameter has a significant impact over the initialization time, although the results obtained are acceptable to achieve a good user experience (they are lower than 1s). Regarding the results obtained for the three algorithms, the number of services included in each Composed Services does not have impact over the execution time of Algorithms a d 2. This is because these algorithms depend only on the number of followed users and the number of Composed Services that each of them created. In contrast, Algorithm 3 needs to access the services included in each Composed Service to find the service that follows a given one. Thus, the size of the Composed Service has a direct impact on its execution time. The way in which execution time changes depending on the number of included services is shown in Figure 5.

Table 3. Initialization time and execution time of each algorithm in experiment 3

# includ. services	Initialization	Algorithm 1	Algorithm 2	Algorithm 3
5	150.16	2.28	2.72	3
10	391.88	2.54	2.8	5.67
15	524.22	2.48	2.91	7.77
20	665.57	2.39	2.58	9.25
25	746.06	2.51	3.05	10.58

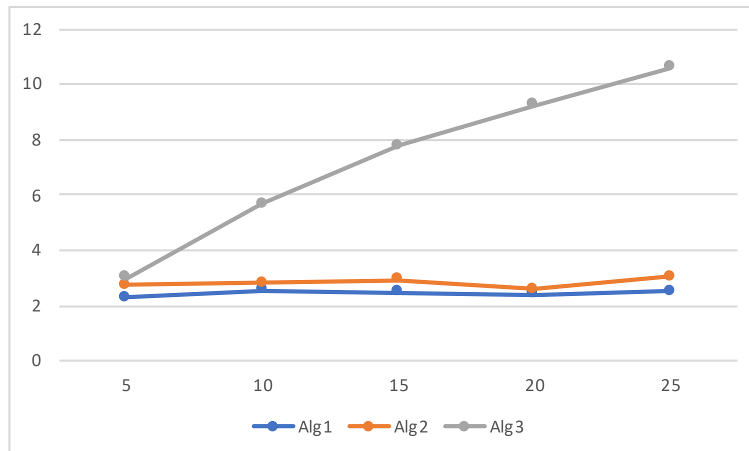

Figure 5. Execution time of each algorithm (in milliseconds, y-axis) for different amount of services included in each Composed Services (x-axis)

Conclusions. The software infrastructure that implements that three proposed algorithms has an acceptable execution time to achieve a good user experience when using the authoring environment. The initialization time is the aspect that more time needs, although it is less than 1 second in every experiment, which introduces little impact on the provided user experience. Note also that we have created an initial data set based on averages calculated from different statistics. This means that we have considered values higher than average, which probably do not occur in a real scenario. For instance, the initial data set considers Composed Services made up of 5 services because this is the average calculated from the data about mashups available in the Programmable Web repository. This means that half of the mashups are made of less than 5 services, which lead us to think that it is little probably that users create a Composed Service that include 25 services.

8 Conclusions

In this work, we have presented a social network to support end-users in the composition of services. We want to encourage end-users to become into producers of services and contribute to improve the research of end-user service composition.

We have characterized the proposed social network and have defined it in a semi-formal way by using graph theory. We have also analysed how social connections can be exploited to (1) facilitate end-users to discover services through browsing these connections, and (2) recommend services to end-users during the composition activity.

References

Aghaee, S., Pautasso, C. (2014). End-user development of mashups with naturalmash. Journal in Visual Language and Computing, 25 (4): 414–432.

Al-Masri, E. & Mahmoud, Q.H. (2007) WSCE: A Crawler engine for large-scale discovery of Web services. In IEEE International Conference on Web Services (ICWS), (pp.1104-1111). IEEE.

Amir, R., & Zeid, A. (2004). A UML profile for service-oriented architectures. In Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (pp. 192–193). ACM.

Boyd, D.M., & Ellison, N. B. (2007). Social network sites: Definition, history, and scholarship. Journal of Computer-Mediated Communication, 13(1), 210-230.

Danado, J., & Paternò, F. (2014). Puzzle: A mobile application development environment using a jigsaw metaphor. Journal of Visual Languages & Computing, 25(4), 297-315.

Daniel, F., Casati, F., Benatallah, B., and Shan, M.C. (2009). Hosted universal composition: Models, languages and infrastructure in mashart. In Conceptual Modeling-ER 2009 (pp. 428–443). Springer Berlin Heidelberg.

De Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., & Taentzer, G. (2007). Attributed graph transformation with node type inheritance. Theoretical Computer Science, 376(3), 139-163.

Ding, Z., Xiao, L., & Hu, J. (2008, October). Performance analysis of service composition using ordinary differential equations. In 12th IEEE International Workshop on Future Trends of Distributed Computing Systems (pp. 30-36). IEEE.

Dogtiev, A. (2018). Instagram Revenue and Usage Statistics (2018). Online available at: http://www.businessofapps.com/data/instagram-statistics/ last time accessed: January 2019.

Ehrig H. (1979). Introduction to the algebraic theory of graph grammars - a survey In Proceedings International Workshop on Graph Grammars and their Application to Computer Science and Biology, Springer-Verlag, pp. 1-69.

Ehrig, H. and Mahr, B. (1985). Fundamentals of Algebraic Specifications 1: quations and Initial Semantics. Vol. 6 of EATCS Monographs on Theoretical Computer Science. Springer.

Ermagan, V., & Krüger, I. H. (2007). A UML2 profile for service modeling. In Model Driven Engineering Languages and Systems (pp. 360–374). Springer Berlin Heidelberg.

Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures, Ph.D. dissertation, University of California, Irvine.

Heckel, R. (2004). Graph transformation in a nutshell. In Proceedings of the School on Foundations of Visual Modelling Techniques (FoVMT) of the SegraVis Research Training Network.

Hung, P.C.K.; Li, H.; and Jeng, J-J. (2004). WS-Negotiation: An overview of research issues, in Proceedings of the 37th Hawaii International Conference on System Sciences (HICSS'04), Big Island, Hawaii, January, pp. 33-42.

Huo, L., Wang, Z. (2016). Service composition instantiation based on cross-modified artificial Bee Colony algorithm. Chin. Commun. 13 (10), 233–244.

IFTTT. (2015). If This Then That. Accesible at: https://ifttt.com/. Last time accessed: January 2019.

Klusch, M., & Sycara, K. (2001). Brokering and matchmaking for coordination of agent societies: A survey. In Coordination of Internet Agents (pp. 197–224). Springer Berlin Heidelberg.

Liberman, H., Paternò, F., Klann, M., Wulf, V. (2006). End-user development: an emerging paradigm. H. Liberman, F. Paternò, V. Wulf (Eds), End User Development, Vol 9. 427-457

Löwe M. (1993). Algebraic approach to single-pushout graph transformation. In Theoretical Computer Science, Vol. 1. pp. 181-224.

Maaradji, A., Hacid, H., Daigremont, J., & Crespi, N. (2010, May). Towards a social network based approach for services composition. In Communications (ICC), 2010 IEEE International Conference on (pp. 1-5). IEEE

Maaradji, A., Hacid, H., Skraba, R., Lateef, A., Daigremont, J., & Crespi, N. (2011, July). Social Discovery and Composition of Web Services. In EUD4Services Workshop-Empowering End-Users to Develop Service Based Applications; Torre Canne, Italy, June 2011.

Milanovic, N., & Malek, M. (2004). Current solutions for web service composition. IEEE Internet Computing, 8(6), 51-59.

Milicevic, A. K., Nanopoulos, A., & Ivanovic, M. (2010). Social tagging in recommender systems: a survey of the state-of-the-art and possible extensions. Artificial Intelligence Review, 33(3), 187-209.

Nielsen, J. (2015, december). Tops of 2015: Digital. Media and Entertainment. Available online at: http://www.nielsen.com/us/en/insights/news/2015/tops-of-2015-digital.html Last time accessed: January 2019

Orejas, F. (2008). Attributed graph constraints. In International Conference on Graph Transformation (pp. 274-288). Springer, Berlin, Heidelberg.

Paolucci, M., Kawamura, T., Payne, T. R., & Sycara, K. (2002). Semantic matching of web services capabilities. In the Semantic Web—ISWC 2002 (pp. 333–347). Springer Berlin Heidelberg.

Papazoglou, M.; Traverso, P.; Dustdar, S.; Leymann, F.; Kramer, B. (2006). Service Oriented Computing Research Roadmap. In: Dagstuhl Seminar Proceedings 05462 (SOC).

Santanche, A., Nath, S., Liu, J., Priyantha, B., & Zhao, F. (2006). Senseweb: Browsing the physical world in real time. Demo Abstract, ACM/IEEE IPSN06, Nashville, TN.

Segal, J. (2005, May). Two principles of end-user software engineering research. In ACM SIGSOFT Software Engineering Notes (Vol. 30, No. 4, pp. 1-5). ACM.

Smith, C. (2018). Interesting IFTTTStatistics and Facts. Online available at: https://expandedramblings.com/index.php/ifttt-statistics-and-facts/ last time accessed: January 2019.

Snoonian, D. (2003). Smart buildings. IEEE spectrum, 40(8), 18-23.

Soriano, J; Lizcano, D.; Hierro, J.J.; Reyes, M.; Schroth, C.; and Janner, T. (2008). Enhancing user-service interaction through a global user-centric approach to soa, in ICNS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 194-203.

Steinbock, D. (2005). The mobile revolution: The making of mobile services worldwide. Kogan Page Publishers.

Valderas, P., Torres, V., Mansanet, I., & Pelechano, V. (2017). A mobile-based solution for supporting end-users in the composition of services. Multimedia Tools and Application. 76(15): 16315-16345.

Wellman, B., & Berkowitz, S. D. (Eds.). (1988). Social structures: A network approach (Vol. 2). CUP Archive.

Workflow.is (2018). Workflow. Spend less taps, get more done. Accessible at: https://workflow.is/. Last time accessed: January 2019.

Yu, J., Sheng, Q. Z., Han, J., Wu, Y., & Liu, C. (2012). A semantically enhanced service repository for user-centric service discovery and management. *Data & Knowledge Engineering*, *72*, 202-218.

Zapier (2018). Connect Your Apps and Automate Workflows. Accessible at: https://zapier.com/. Last time accessed: January 2019.