2012 IEEE 16th International Enterprise Distributed Object Computing Conference

A-DynamiCoS: a Flexible Framework for User-centric Service Composition

Eduardo Gonçalves da Silva, Luís Ferreira Pires, Marten van Sinderen
Faculty of Electrical Engineering, Computer Science and Mathematics
University of Twente
Enschede, the Netherlands
e-mail: {e.m.g.silva, l.ferreirapires, m.j.vansinderen}@utwente.nl

Abstract *— Service composition has been acknowledged as a promising approach to create new (composite) services that are capable of supporting multiple needs of service users. Service composition has been used quite extensively to support complex but relatively stable enterprise processes. More recently, service composition is also being applied to support the personalization of services delivered to human end-users. In these situations, the aim is to create service compositions on demand, at runtime, that match the specific requirements of each individual end-user, characterizing user-centric dynamic service composition processes. The most common approach to dynamic service composition aims at creating service compositions that once obtained must be deployed and executed by some sort of orchestration engine. In this approach, the service composition life-cycle is fixed, with serious limitations to its applicability in more realistic situations. This paper presents the A-DynamiCoS framework, which applies a novel approach to user-centric service composition by providing flexible support to the different activities of the service composition life-cycle. In this approach, components that perform service discovery, composition and execution activities are orchestrated, breaking the traditional rigid life-cycle (discover, then compose, then deploy and finally execute). We claim that dynamic orchestration of service composition activities enables true user-centric service delivery. This paper motivates and justifies the development of A-DynamiCoS, and presents use cases that demonstrate our claims.*

Keywords - service composition; user-centric; service-oriented architecture

I. INTRODUCTION

Service composition has been acknowledged as a promising approach to create new (composite) services that are capable of supporting the needs of service users [1]. A potential benefit of service composition is that it allows new services to be created rapidly, as a combination of existing basic services, instead of being developed from scratch. Service composition is being used extensively to support complex but relatively stable enterprise processes with the Business Process Management (BPM) systems employed by many business organizations. However, service composition is also being applied to support the personalization of services delivered to human end-users. In these situations, the aim is to create service compositions on demand that match the specific requirements of each specific end-user. We define this process as *dynamic service composition* [2].

Many efforts to perform dynamic service composition have been reported in the literature. We refer to [3] for a comprehensive account of (dynamic and automatic) service composition approaches. The most common approach to dynamic service composition assumes the generation and representation of service compositions to satisfy a specific set of requirements, which once obtained should be deployed and executed by some sort of orchestration engine, commonly WS-BPEL [4] engines. In this approach the service composition life-cycle is fixed: services are discovered, composed, then deployed so that service execution can take place.

However, this fixed service composition life-cycle poses serious limitations when not all the service (composition) requirements are known beforehand. For example, consider a user planning some leisure activities by using multiple basic web services, e.g., to find restaurants, find hotels, find routes, etc. In this example, a fixed life-cycle for service composition would create two main problems: (1) not all the users require exactly the same composition of services (e.g., book hotel then book restaurant), and (2) often users do not know all their requirements (and the necessary services) beforehand. A user may decide what services should be used next, only after the execution of a given service. For example, a service for finding a route to a hotel would become necessary after learning that the hotel that was booked is in an address unknown by the user.

In order to address these problems, we have developed a *user-centric service composition* framework, which we call A-DynamiCoS (Adaptable-DynamiCoS). This framework applies a novel approach to service composition by providing flexible support to the different activities of the service composition life-cycle, according to the specific needs of the users at a given moment. In this approach, components that perform service discovery, composition and execution activities are orchestrated, breaking the traditional rigid life-cycle (discover, then compose, then deploy and finally execute). We claim that dynamic orchestration of service composition activities enables true user-centric service delivery. This paper motivates and justifies the development of A-DynamiCoS, and presents use cases that demonstrate our claims.

This paper is further structured as follows: Section II motivates and gives an overview of the A-DynamiCoS architecture, in terms of its components and adaptable coordination; Section III discusses the user support offered

1541-7719/12 $26.00 © 2012 IEEE
DOI 10.1109/EDOC.2012.19

by A-DynamiCoS to enable user-driven composition of services; Section IV discusses the coordinator component that is responsible for orchestrating the basic components of the framework as a function of requested user commands; Section V briefly discusses two use cases in different application domains, which illustrate the general applicability of the framework; Section VI discusses related work and Section VII gives our conclusions.

II. A-DynamiCoS Architecture Overview

A system that supports user-centric service composition has to shield the end-user from the details of the service composition process. Often service end-users do not have the technical skills to handle the service composition process, and the service composition process is supposed to be performed on demand (at runtime), whenever the user requires some service (composition). Service composition details can be (partly) hidden from the user by automating the service composition process, as we have demonstrated in [5] with the DynamiCoS framework [1]. Similarly to most dynamic service composition approaches, this framework assumes that the user formulates a service request with all the service (composition) requirements in a single interaction. This service request is then forwarded to the supporting system to trigger the composition process. This assumption ignores the possible differences in domain knowledge and circumstances of different users, which limits the applicability of the framework to realistic situations. For example, if the user is unfamiliar with the application domain, she may not be able to specify all her requirements and needs more assistance when defining the service request [6]. Furthermore, even if the user knows the application domain, often the identification of requirements or a decision to invoke a service only takes place after executing some services. For example, the need to execute a service to find a route to a hotel could be triggered once the user realizes that she does not know that specific part of the city where the hotel is located.

By acknowledging that users are heterogeneous (have different knowledge and skills) and change or identify their requirements as they advance in the service composition process, we developed the A-DynamiCoS framework, as an extension of the DynamiCoS framework. For this extension we have separated the DynamiCoS basic components (shown in Fig. 1) into independent components, in order to invoke them in different orders. We have defined an adaptable Coordinator, which invokes these basic components according to the characteristics and requirements of the user driving the service composition process. The Coordinator receives different commands from the user, and reacts on-demand to the user's requirements, without having any predefined (fixed) service composition life-cycle. The DynamiCoS basic components and the A-DynamiCoS extensions are described below.

[1] http://www.dynamicos.sourceforge.net

A. DynamiCoS Components

In DynamiCoS, each phase of the service composition life-cycle is supported by a component. The components are integrated in a flow that encompasses the traditional dynamic service composition life-cycle [2]: (1) the user expresses his request in terms of goals; (2) services that can potentially be used in the service composition are discovered; (3) discovered services are composed to match the user service request; (4) the resulting service composition is deployed, so that it can be executed by the user.

Fig. 1 gives an overview of the DynamiCoS architecture, showing its basic components.

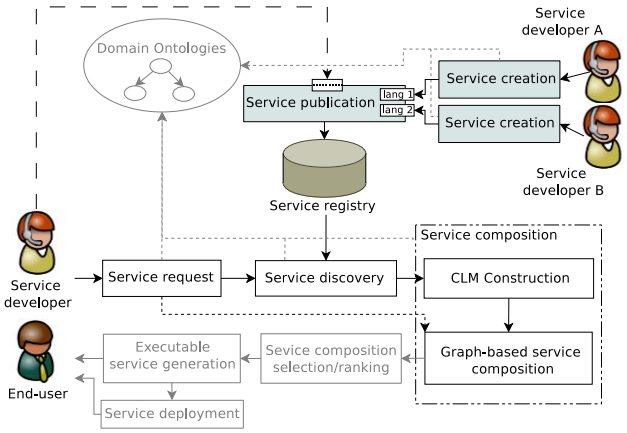


Figure 1. DynamiCoS architecture and basic components.

The DynamiCoS framework assumes that the user is able to specify a detailed set of requirements in one interaction, i.e., it assumes that the user is familiar with the domain in which she is seeking services.

The DynamiCoS basic components have been originally defined to automatically support the whole service composition process, aiming at shielding users from the service composition details. Automation is achieved by using semantic information to describe services, user requests and in the discovery and composition processes. A component has been defined for each activity required to support the service composition life-cycle phases, such as the *ServiceDiscoverer* for service discovery and the *ServiceComposer* for service composition. Furthermore, we have developed auxiliary components that support some activities of these two components, namely the *SemanticReasoner*, which performs semantic reasoning on services parameters and interface semantic concepts based on ontologies, and the *CLMManager*, which manages the CLM matrix [7, 8].

The composition framework components are stateless, i.e., they perform a given set of operations and do not keep any state. The state of the service composition is kept in a set of data structures, which hold the discovered services and service compositions being created. A service has a service interface with inputs and outputs parameters, and other properties, namely goals, non-functional properties and its natural language description. The CLM stores all the possible semantic links between the outputs and inputs of the

discovered services. Furthermore, the service composition, which is represented in the *ServiceComposer* as a graph, is composed of multiple services.

B. A-DynamiCoS

Although the DynamiCoS framework defines a rigid workflow of activities, it has been built by composing the basic components necessary to address the different phases of the service composition life-cycle. By decoupling these components from the DynamiCoS rigid workflow, we could use them to build flexible support to the user-centric service composition process. The extension developed consists of introducing a flexible coordination that invokes the necessary basic components as a function of the user requirements and characteristics at each step of the service composition process. Fig. 2 shows the architecture of the A-DynamiCoS framework [9].

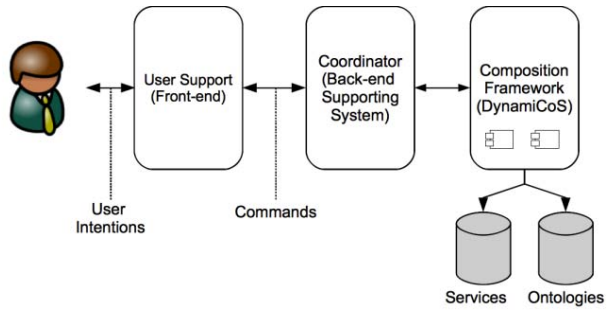


Figure 2. Extension for adaptability.

The A-DynamiCoS framework consists of three components, namely the *Composition Framework*, the *Coordinator* and the *(Front-end) User Support*. The aim of the user support component is to mediate and collect the user's intentions at each step of the composition process, shielding the user from the actual details of the service composition process. To accomplish this, the user support component translates the user intentions at each step of the service composition process to commands that indicate which activities have to be performed. For example, the user support can be implemented as a simple web page that presents the user with a search box where she can describe in natural language the services she is looking for. A request on this web page can be mapped onto a service discovery command, which instructs the Coordinator to carry out a service discovery activity to find services that deliver the requirements specified by the user.

The Coordinator component is responsible for receiving the commands from the User Support component, and for mapping them onto the necessary invocations of the basic components of the composition framework, which deliver the required behavior. In the example above, whenever the coordinator receives a command to perform service discovery, it invokes the ServiceDiscoverer component, which queries the service registry for services that match the requested parameters. The matching services are then returned to the User Support as result.

The User Support component, which basically defines a user interface, is defined as a function of the characteristics and requirements of the target user population to be supported, and the application domain in which services are to be composed. In contrast, the Coordinator and the Composition Framework are generic and can be reused in different application domains to support different types of users. Therefore, the basic components of the Composition Framework provide the required support for the service composition process in different situations in different domains, possibly with different users. Only the order in which the basic components are used varies.

III. USER SUPPORT

In A-DynamiCoS we have defined *commands* to instruct the Coordinator (back-end supporting system) to execute the different activities of the service composition process. These commands are issued by the (front-end) User Support component to support end-user interactions.

A. Commands

Each command encapsulates a request for a given behavior to be delivered by the basic components of the composition framework. Commands are encoded in messages, which are interpreted by the Coordinator. We followed the command pattern [10] in this architecture in order to handle different primitive commands that can be issued by the User Support. The command pattern gives a structured way to encapsulate different required behaviors in a single message format.

The service composition life-cycle supported by DynamiCoS (shown in Fig. 1) consists of service request, service discovery, service composition and service execution activities, performed in this specific order. In realistic user-centric service composition situations, two different users may need to perform these activities in different order, according to their requirements.

Fig. 3 presents two execution workflows, where two different users drive the service composition processes differently.

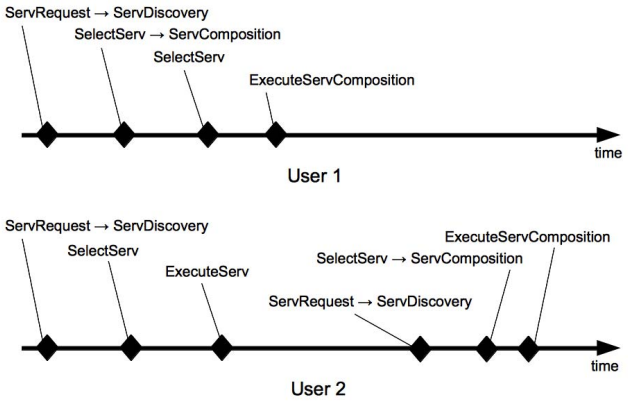


Figure 3. User Command workflow examples.

User 1 follows the order determined by the fixed life-cycle. User 2 performs the service composition activities in the following order:

1. The user specifies a service request, defining the requirements for a requested service.
2. The service discovery activity takes place, where services that satisfy the requirements are discovered.
3. The user selects one of the services.
4. The user is presented with an interface to use the service (service execution activity).
5. Based on the result, the user decides to use another service, so she requests a new service (service request).
6. The service discovery activity takes place again to find services that can fulfill the new requirements specified by the user.
7. The user selects one of the discovered services.
8. The user is presented with an interface for using this new service (service execution activity). In this step some input values can be automatically entered, or suggested to the user, because they are available from the execution of the first service. This process can go on, as long as necessary.

Although the two users have different workflows of supporting activities, we can observe that they use the same basic activities, namely: service request, service discovery, select, service composition, and service execution. This shows that we can define generic commands to support the different activities of the service composition process in different situations, where users with different requirements can be seamlessly supported by the same basic commands. Although the same set of commands can be used in different order and across multiple situations to support different users, the order in which the commands can be invoked has to comply with some rules. For example, a service execution command can only be issued after a service has been discovered. To describe these interdependencies between commands we define a *dependency graph* [9] of command types. The dependency graph specifies the basic types of commands and their causal relations.

We refer to the commands that can be issued to the back-end supporting system coordinator as *primitive commands*. A primitive command defines a usage protocol, specifying its request and response messages. The command type of a primitive command defines its preconditions or dependencies on other primitive commands. It is not reasonable to assume that the supporting system designer will envision at once all the possible primitive commands to support any possible usage scenario. Therefore, the supporting system has to be extensible concerning the definition of new primitive commands, i.e., it should be possible to add new primitive commands in the supporting system later on. A-DynamiCoS allows designers to add new primitive commands as long as they comply with the dependency graph. This guarantees that newly added primitive commands do not introduce undesired behaviors, which may lead to inconsistency.

We have initially identified several primitive commands mostly directly related with the service composition life-cycle activities, such as:

- *ServTypeDiscovery*: discovers a service based on a given (semantic) type;
- *SeleServ*: selects a service and adds it to a service composition;
- *ExecServ*: retrieves service composition services ready for execution;
- *ValidateInputs*: stores the service inputs entered by the user if they are valid;
- *AddToContext*: stores user context information or service results in the execution context of the back-end supporting system.

B. *User interactions*

The front-end User Support component defines the interface between the users and the back-end supporting system. Fig. 4 shows the internal structure of the front-end User Support component. It consists of two parts, namely the User Interface and the Command Flow. The User Interface defines the interaction points for the supported application, where the user intentions concerning the service composition process are collected. The Command Flow defines workflows of primitive commands defined to support the identified target user population to be supported. This component translates user intentions to primitive commands understandable by the back-end supporting system, allowing the user to drive the service composition process.

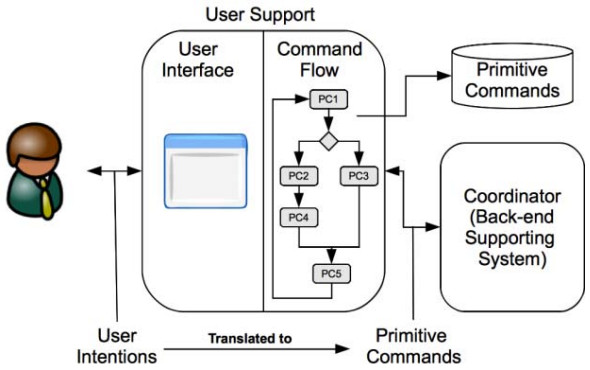


Figure 4. A-DynamiCoS front-end User Support.

Because the User Support should capture the essential activities that can be performed in a given service composition usage scenario, the User Support should be defined by domain specialists, taking into account the characteristics of the application domain and the target user population of the usage scenario.

A Command Flow is defined as a combination of primitive commands organized in a workflow suitable to fulfill the requirements identified for a given application. The dependency graph is defined to guarantee that command flows respect the command types interdependencies, providing in this way the basic rules for defining valid command flows. Multiple primitive commands can be accessible to the user at a given moment, which allows the user to decide which workflow of activities shall be followed. Although the User Interface has to follow a prescribed Command Flow, the user does not have to be

aware of the existence of this Command Flow, as long as the front-end user support can capture the user intentions at each step of the service composition process.

Fig. 5 shows an example of a Command Flow described as a UML activity diagram [9]. In this Command Flow, a user starts by defining which service she wants to use, which is translated into a *ServTypeDiscovery* command. The user then gets a set of services, from which she selects one through the *SeleServ* primitive command, which communicates the selected service to the back-end supporting system. The user then is presented with the service interface, where the different inputs of the service are requested. The user enters the inputs she knows. Once all known inputs are entered, the *ValidateInputs* primitive command is used, which communicates these inputs to the back-end supporting system as well as which inputs the user cannot provide. In case the user cannot provide some inputs, the Command Flow issues the *ResolveServ* command, which requests the back-end supporting system to discover services with outputs that can provide the missing inputs. This triggers a *backward-chaining* service composition process, in which the user selects the services to be used to provide each of the missing inputs. Once all the services inputs are available, the service composition can be executed, by issuing the *ExecServ* command. One service is executed at a time in this process.

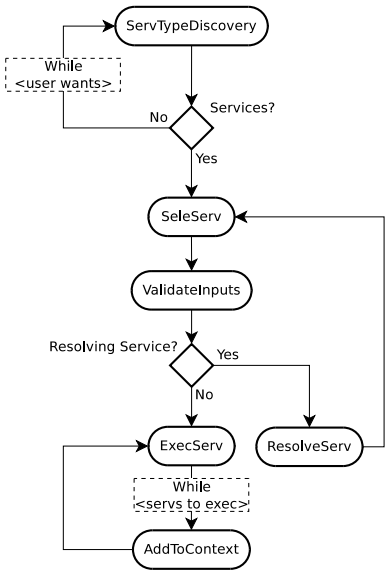


Figure 5. Command Flow example.

IV. COORDINATOR COMPONENT

The Coordinator is the central component of A-DynamiCoS. It consists of sub-components that handle commands (Interpreter, Validator and Executor) and the Composition and Execution Context, which basically holds the state of each user session being handled at each moment, allowing multiple user sessions to be supported simultaneously.

Fig. 6 shows the Coordinator internal structure and its relation with the other components of the composition framework.

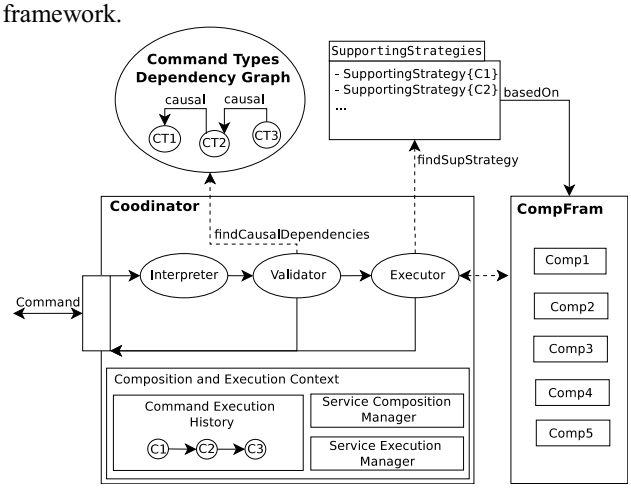


Figure 6. Coordinator internal structure.

A. Handling Commands

The Coordinator gets a command that represents the user's intention, and executes the supporting strategy to deliver the behavior required by the user. Supporting strategies are defined in terms of invocations of basic components of the composition framework in a given order. The coordinator has been defined to be generic and extensible, i.e., it is independent of the primitive commands being processed, the supporting strategies that are executed and the composition framework components that are called to realize the supporting strategies. Furthermore, the Coordinator enforces the soundness of the coordination being performed, by checking the order of execution of primitive commands against their dependencies, through the dependency graph.

Each primitive command is represented in a message with three parameters: (1) *UserSessionID*, which identifies the user session being processed; (2) *CommandID*, which identifies the type of primitive command issued to the coordinator; (3) *CommandParameters*, which contains the parameters of this specific primitive command. Once the coordinator receives a message containing a primitive command, this message is processed by the Interpreter, which parses the message to extract the message parameters, and opens the user session context associated to the *UserSessionID*. The user session context is obtained from the Composition and Execution Context.

Once a message is interpreted, the Coordinator proceeds to perform validation. The Validator verifies whether the issued interaction type is valid by inspecting the dependency graph for the causal dependencies of the issued primitive command. To achieve this, it inspects the command execution history, where the primitive commands previously issued by the user are stored. In case the primitive command is invalid (i.e., it disobeys some causal dependency), the coordinator aborts the execution of its supporting strategy, and returns an error message to the front-end user support.

B. Composition and Execution Context

The Composition and Execution Context component keeps the data handled in the service composition process, namely, the information values from services' inputs and outputs and information from the user's context that can be used as inputs for services. Fig. 7 shows the data structures and components of the Composition and Execution Context (*Command Execution History*, *Service Composition Manager* and *Service Execution Manager*).

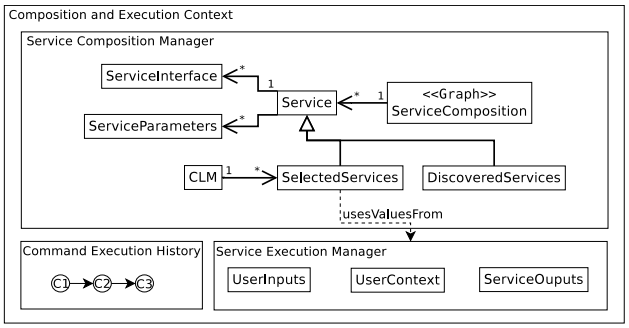


Figure 7. Data structures and components of the Composition and Execution Context.

The Command Execution History is responsible for keeping track of all the primitive commands issued by the front-end user support and executed in each user session. This component allows the Coordinator to verify the correct order of execution of the primitive commands, guaranteeing in this way that the supporting system does not enter inconsistent states.

The Service Composition Manager is responsible for storing the service compositions being created. This component also uses the class *SelectedServices* to store the services that the user selects for composition and execution from the list of *DiscoveredServices*. *SelectedServices* are used to define the *CLM* (Causal Link Matrix), which is a matrix that contains all the candidate services that can be used in service compositions. Based on these data structures, the supporting strategies can perform the different activities required to support users in the service composition process. These data structures maintain the state of an instance of the composition process (user session), while the basic composition framework components are used to perform the necessary processing in the service composition process, such as, e.g., adding a service to a service composition or discovering a new service.

The Service Execution Manager is responsible for keeping the information values associated with the user and services execution context. It stores data values for user inputs, user context (values gathered from the user's context, such as, e.g., user location) and service outputs. We argued before that most service composition approaches separate quite explicitly service composition creation from service composition execution. In these approaches, the composition execution engine manages all the service parameter values at runtime. Since we do not explicitly separate service composition creation from service execution, in our user-centric approach, the Service Execution Manager is integrated in the back-end supporting system. This design decision allows the framework to support service composition processes that alternate between the composition and execution of services, since it allows the framework to keep track of the service parameter values, while further changes on the service composition are still possible.

C. Service Composition and Execution Process

In order to support true user-centric service composition scenarios, we have to create service compositions for each specific end-user, with a specific set of requirements, as opposed to most service composition approaches in the literature. Therefore we need to cope with the heterogeneity of end-users, as indicated before. Many users have some initial requirements, and require assistance to formulate them properly. They may have to learn about the application domain of the services being composed, so that they can take decisions and proceed in the service composition process. Their limited knowledge may imply that they have to use auxiliary services in order to learn about the domain while the intended service composition is being built. This gives an additional role to the service composition environment.

Fig. 8 shows a simple example of workflow that can be supported by our user-centric service composition approach:

1. The user looks for a service to satisfy requirement R1, for which S1 is discovered.
2. The user selects S1.
3. The user executes S1.
4. Once the user executes S1, she obtains some new information ('learns') and decides that she needs another service to satisfy a new requirement R2 that she has just identified/formulated.
5. S2 is discovered and selected by the user.
6. S2 can reuse information from S1 inputs/outputs, i.e., S1 is composed with S2, resulting in the composition S1S2.
7. The resulting composition (S1S2) is then selected for execution.
8. S1S2 is executed, which corresponds to the execution of S2 by reusing some values from the S1 inputs/outputs.

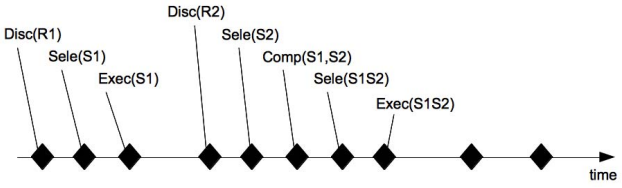


Figure 8. Example of user-centric activities flow.

This simple example shows that user-centric service composition is a dynamic process, and the service composition can be defined and extended dynamically, possibly leading to workflows where execution and composition of services are interleaved. We have taken an architectural design decision to define service compositions in an incremental fashion so that services can be executed while the service composition is still being constructed or

extended. Therefore we do not perform service composition deployment, but assume that services from the service composition are executed one at a time, whenever they are ready for execution and have the required information values to be executed. To cope with this, we delegate the actual invocation of execution of services to the front-end User Support component, while the back-end supporting system manages all the information values required for service execution and the results of service execution.

Service results are stored in the Composition and Execution Context component, which can be used later by other services. For example, in Fig. 8, the results from S1 can later be used in the execution of S2. Apart from allowing a more flexible composition and execution of services, this architectural decision allows us to make the back-end supporting system generic and independent of concrete service execution technologies. The drawback of this architectural decision is that the front-end User Support has to handle the process of invoking the execution of services. Nevertheless, this decision is justifiable since the front-end User Support is application domain-specific, allowing us to define mechanisms to execute services that are specific for the application domain to be supported. Furthermore, this decision is beneficial because services can be implemented with different application transport mechanisms (e.g., RESTful web services [11], SOAP web services [12] or even proprietary APIs), and these different mechanisms are encapsulated in the application-specific front-ends.

D. Supporting Strategies

Supporting strategies are defined to realize the different primitive commands as functions of the basic components of the composition framework and the Composition and Execution Context data structures. Supporting strategies are defined to invoke the different composition framework components on demand, as they are required by the users being supported in the service composition process. Furthermore, the supporting strategies make use of the Composition and Execution Context data structures to store and handle services and service compositions, and information associated to services execution and user context. New components can be added to the composition framework, or new primitive commands may be required. This allows one to define new supporting strategies that implement new behaviors whenever necessary.

We have designed A-DynamiCoS to allow these extensions, while keeping the same dynamics for the Coordinator, namely to interpret primitive commands, validate them, and then execute a supporting strategy to implement the support required for each primitive command. This makes A-DynamiCoS framework flexible, extensible and adaptable to new requirements. The definition of supporting strategies can be made without having to know the behavior of the Coordinator. The designer of primitive commands and their supporting strategies only needs to know the composition framework components, their interfaces, and the Composition and Execution Context data structures that hold the different information handled by the supporting strategies.

Fig. 9 shows the example of the supporting strategy we defined for the *ServTypeDiscovery* primitive command, which allows the discovery of all services of a given type. The basic actions taken in this supporting strategy are:

1. Discover matching services for each requested service type (ServType) using the ServiceDiscoverer composition framework component.
2. In case services have been discovered, add the services to the Composition and Execution Context of the user session instance, more specifically to the *Discovered-Services* data structure.
3. Return a list (possibly empty) of discovered services ordered according to the requested types.

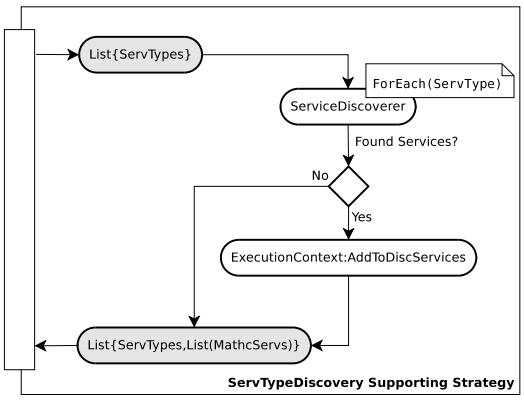


Figure 9. Supporting strategy for *ServTypeDiscovery* primitive command.

V. USE CASES

In order to demonstrate the suitability of A-DynamiCoS to support user-centric dynamic service compositions, we implemented a prototype and applied this prototype to two use cases in different application domains, performed in two separate Master projects [13, 14]. The use cases focused on developing front-end User Support components, which make use of the back-end supporting system (Coordinator and Composition Framework) prototype.

A. Back-end Supporting System Implementation

In the A-DynamiCoS prototype, primitive commands are defined using XML Schema Definitions (XSD). The primitive commands XSD define the request and response messages to be exchanged. Based on these definitions, the front-end user support designers (domain specialists) can create the request messages for each primitive command used in the front-end user support. These definitions are also used in the A-DynamiCoS coordinator to generate the necessary functionality to handle the messages of each primitive command, i.e., to process the received messages and then to create the response messages according to the primitive commands issued. Since our prototype was mainly developed in Java, it was easier to first define the XSD and then generate automatically the code to unmarshal the primitive command request XML messages and marshal their response XML messages. We used tooling that supports JAXB in order to generate this code.

The A-DynamiCoS back-end supporting system was implemented in Java, and exposed as a web service with a single operation to get primitive commands. Supporting strategies are implemented as Java methods of the Coordinator Class. The methods are called when a primitive command is received, immediately after its validation. All the supporting strategies share a common structure for their workflow of activities:

1. Unmarshal the parameters from the primitive command message.
2. Perform the supporting strategy, as a combination of calls to the DynamiCoS basic components.
3. Marshal the response message to be sent back to the front-end user support.

The DynamiCoS basic components reused in A-DynamiCoS are implemented in Java and exposed as Java classes that can be used in the definition of the supporting strategies.

The Composition and Execution Context component has been implemented as a list of objects that store the different elements of the user session: service compositions, discovered services, execution context (values), the user's context and the interaction execution history. Each time a new user session is initiated, a new composition and execution context is created.

B. E-Government Use Case

The e-government use case [13, 15] defines a usage scenario that aims at facilitating the access of citizens to government electronic services. This specific usage scenario is expected to benefit from user-centric service composition approaches to attend specific requirements from citizens, since different citizens, have different characteristics and requirements for the public services to be used.

In this use case, we developed a simple web portal for users who want to access government services. This portal concentrates different public services from different departments of a government into one place where users can be assisted to make use of these services. Users (citizens) normally have limited knowledge on the e-government application domain, and have limited information on the services that are available in the domain, since this domain is large, with many services and bureaucratic processes. To cope with this, the supporting system has to assist users to discover services and resolve the preconditions of the services that they want to use. Furthermore, since users may have very limited technical skills, they require very simple and intuitive interfaces to interact with the system, invoke services, communicate problems, request information from the system, or request assistance to resolve the preconditions of the selected services.

Because of the users characteristics mentioned above, we have defined the following workflow of activities to support citizens in the process of finding and using public electronic services:

1. A user specifies her *goal service* in a simplified interface, which collects the type of service the user needs.
2. The system retrieves possible services that match the service type the user requested.
3. The user gets information concerning the discovered services, which allows her to select a service for usage.
4. In case the user selects a service, she gets further information on the preconditions and/or inputs required for using the service. If the user does not know some of the preconditions or inputs, she may request assistance to resolve them. This assistance finds services with outputs that provide the unfulfilled preconditions and/or inputs as output. This process can be repeated several times if necessary.
5. Once the user resolves all the preconditions and inputs of the goal service, she can make use of the service.

Fig. 10 shows the Command Flow defined to support this scenario.

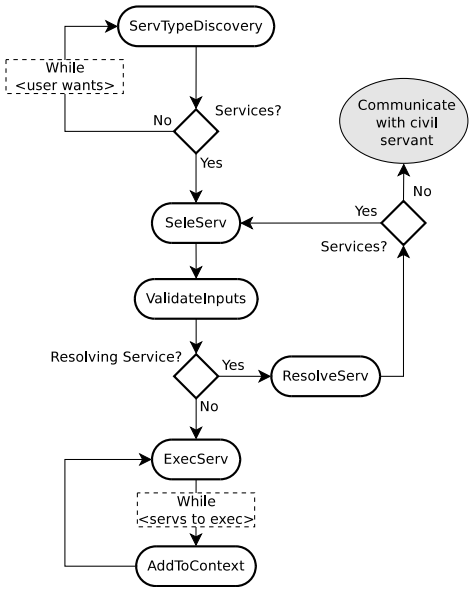


Figure 10. Command Flow of the E-Government use case.

This Command Flow uses the primitive commands in the following way:

1. *ServTypeDiscovery*: discover services of a given type. If no services are found, asks the user to refine request.
2. *SeleServ*: selects a service for execution. Furthermore, at the back-end supporting system, the selected service is added to the service composition.
3. *ValidateInputs*: after the user selects a service, she is informed about the preconditions and inputs required to use the service, and she is asked to provide the necessary inputs. The ValidateInputs command communicates the information entered by the user to the back-end supporting system, which is stored for later usage when the service is to be executed. The preconditions and inputs that the user does not know are set as unfulfilled, which means that they still need to be entered by the front-end user support or resolved by another service output.

4. *ResolveServ*: in case there are unfulfilled inputs the ResolveServ command is issued, discovering services that have outputs that semantically match the unfulfilled pre-conditions or inputs. If services are found, they are communicated to the user, which issues a SeleServ command to indicate which service is to be considered for resolving the unfulfilled precondition or input. In the back-end, the SeleServ command triggers a composition between the unfulfilled service and the selected service, which corresponds to a *backward-chaining* service composition process. In case no services are found, the user is directed to a civil servant that may help her find the missing information. The process of resolving the preconditions and inputs of a service recurs until all the services in the composition have their preconditions and inputs available, so that all the services of the composition can be executed.
5. *ExecServ*: this command queries the back-end supporting system for the next services in the service composition that are ready to be executed, i.e., the services from the service composition that have all the precondition and input values available. A-DynamiCoS delegates the actual execution of the services of the service composition to the front-end User Support, although the back-end manages the services that are ready for execution, and the keeps the results (output information) of these services. The front-end should be capable of invoking the different services of a service composition.
6. *AddToContext*: once a service is executed in the front-end user support, the resulting values are reported to the back-end supporting system through the AddToContext primitive command. The AddToContext primitive command changes the status of the executed service to "executed", and stores the service output results in the Composition and Execution Context. After that, the ExecServ command is invoked again to request the next service, which can be a service that uses an output of the service that was executed in the previous iteration. The ExecServ primitive command is issued again until all the services of the service composition are executed.

In the use case, this command flow has been used to implement an automated procedure for a handicap citizen to order a parking place nearby his house, which has been extensively reported in [13, 15].

C. *Entertainment Use Case*

The Entertainment use case [14] aims at assisting users to plan leisure activities. This use case benefits from user-centric service composition because of the different requirements different users have at distinct moments when looking for or planning leisure activities. In this use case, users are presented with a web interface where multiple services can be used to plan different activities and also obtain diverse information concerning those activities and other related events. We assume that users have limited knowledge on the entertainment application domain and its services. We also assume that users have limited technical skills. Furthermore, we assume that users can have multiple goals, which can change overtime as users make use of services in the domain. For example, a user may use a service to look for cultural activities at a given location, and since the event finishes at dinner time, she can also decide, at that moment and based on the information she learned about the time and location of the activity, to book a table in a restaurant nearby the place where the activity takes place. This simple example shows that user goals may evolve overtime as the user makes use of services.

For this use case we omit the detailed discussion of the workflow of activities and discuss the Command Flow directly. Fig. 11 shows the Command Flow defined to support this usage scenario.

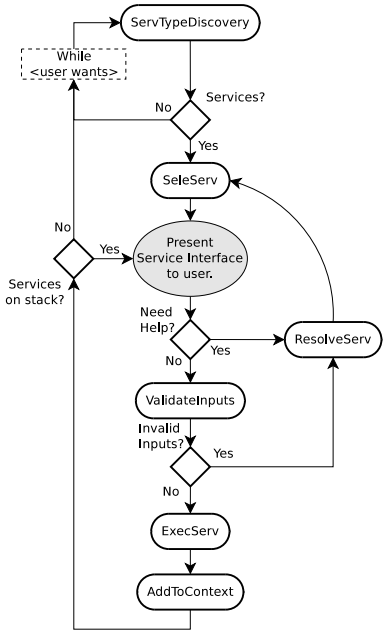


Figure 11. Command Flow of the Entertainment use case.

This Command Flow uses the primitive commands in the following way:

1. *ServTypeDiscovery*: the user starts by making use of the service discovery facility, indicating the service she wants (the *goal service*). This is translated into a ServTypeDiscovery primitive command, which instructs the back-end supporting system to discover all the services that match the service type (semantic type) specified in the user request. This results in a list of services, which can be empty in case no services are discovered to match the user service request properties. If no services are returned, the user is informed and redirected to the initial page.
2. *SeleServ*: in case services are found, the user gets a list of services, and some information describing them. Based on this information, the user decides which service better fits her specified needs, and selects one of these services. Once a service is selected, a SeleServ primitive command is issued to the back-end supporting system. This command adds the selected service to the

service composition and the Composition and Execution Context of this user in the back-end supporting system.

3. *Present Service Interface*: the service interface specifies the precondition and input parameters a service requires to be used. For each parameter of the interface, the user may get value suggestions. These suggestions are taken from the user execution context, and may be from inputs and outputs of services previously executed, or from the user's context (e.g., user's location). The user selects suggestions or enters new values for each precondition and/or input of the service interface.
4. *ResolveServ*: if the user is not able to provide some of the service inputs she can request help to resolve them, which triggers the ResolveServ primitive command. This command retrieves all the services that can deliver an output that semantically match the type of the service precondition or input that the user is trying to resolve. The ResolveServ command offers several services. The user selects one service to resolve the precondition or input from the previously selected service, using the SeleServ primitive command.
5. *ValidateInputs*: sends the user entered inputs, or outputs provided by other services retrieved via the ResolveServ primitive command, to the backend supporting system. These values are stored in the Composition and Execution Context component.
6. *ExecServ*: when all the service preconditions and inputs of the service composition are available the Command Flow proceeds to the ExecServ primitive command. This command is used to query the back-end supporting system for the services that can be executed. The returned services are executed one at a time.
7. *AddToContext*: this command informs the back-end system that a service was executed, by communicating the results (outputs) of the service, which are stored in the Composition and Execution Context of this user session.
8. *Initial Supporting Interface*: after executing all service composition services (the goal services), the user returns to the beginning of the Command Flow, which allows her to request new services. The services requested from then on can use the values from the Composition and Execution Context, which means that if a service precondition or input is semantically related to values from the user's Composition and Execution Context, these values are presented as suggestions. This facilitates the reuse of information, relieving the user from keeping track of all the information handled in the whole service composition and execution process and from repeatedly re-entering information.

This Command Flow was used to implement a prototype application that allows users to plan a set of activities. This application was tested with the situation in which the user wanted to arrange a weekend visit to some place in the Netherlands. In this example, the *Kayak* service was used to find hotels, the *Ikdoe* service was used to find cultural activities and the *Iens* and *Seatme* services were used to find restaurants. These services were invoked through their web APIs. These services were composed and executed transparently to the end-user, under the control of the front-end User Support and our back-end Coordinator. Fig. 12 shows a screenshot of the user interface of this application. This use case has been extensively reported in [14].

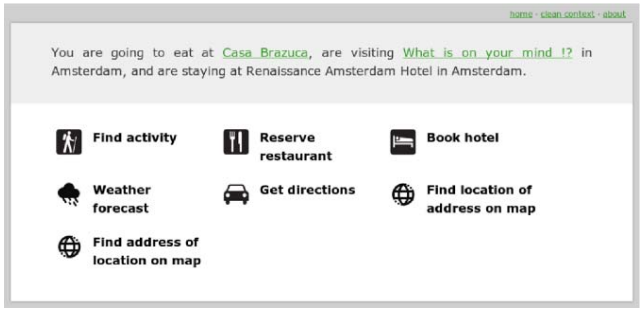
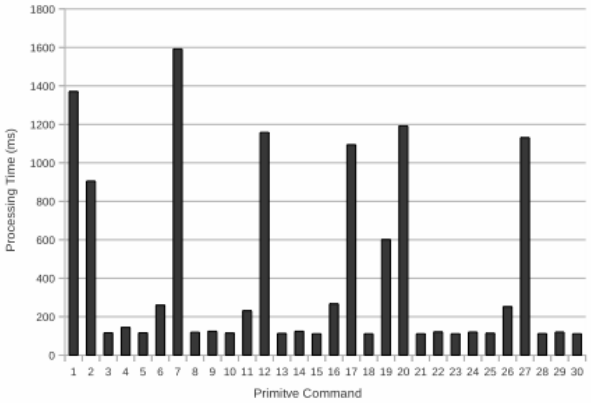


Figure 12. Entertainment Use Case User Interface.

D. *Performance evaluation*

In our experiments we have done some performance evaluation of the prototype, amongst others by measuring the processing time taken to support the different primitive commands issued by the user. The purpose of these measurements has been to get an indication of the applicability of the framework to realistic scenarios.

The setup used consisted of a client and a server machine (Pentium IV, 3.4 GHz, 2GB RAM) connected to the same network in our laboratory. Fig. 12 provides the observed average processing times for the different primitive commands issued on a usage scenario, in this case for the entertainment scenario. The variance of the measured processing times was always very small, lower than 5%.



The primitive commands that took longer to process were SeleServ (2, 7, 12, 17, 20, 27) and ServTypeDiscovery (1, 6, 11, 16, 26). This was expected because these commands require semantic reasoning to be performed, which is an expensive processing task. Nevertheless, these results have shown that the response times are suitable to deliver real-time support to users, which is a major requirement for user-centric dynamic service composition.

VI. Related Work

When surveying the literature we noticed that a large amount of research efforts on user-centric service composition address the development of algorithms to find suitable service compositions for a given set of user requirements. These efforts specially focus on (1) finding all the services the user needs; (2) deploying the resulting service composition. Below we briefly discuss some of the most relevant *dynamic service composition approaches* we came across in our research. In [16] we report on our earlier survey on this topic.

The MoSCoE approach [17] proposes a framework for service composition and execution. Their main distinguishing factor from other approaches is the support of reformulation of the service request. The service composition is performed in three steps: *abstraction*, *composition* and *refinement*. A user request is expressed in a state machine representation, where the user basically specifies functional and non-functional properties for the service to be composed. The composition process is performed using Symbolic State Systems (STS), and is carried out automatically. If there is a service composition that can satisfy the service request, a composite service is returned; otherwise the user is asked to refine the service request, and perform a new iteration in the composition process. Whenever a composite service is successfully created it can be translated to BPEL, allowing the deployment and execution of the service composition.

Kona et al. [18] propose an approach for the automatic composition of semantic web services. Their approach uses a graph-based service composition algorithm. The composition process is performed using a *multi-step narrowing algorithm*. The user specifies a service request, or a query service, specifying the *IOPE* parameters for the desired service. The composition problem is then addressed as a discovery problem, starting by discovering the request inputs and preconditions, and iteratively resolving the *open* outputs and post-conditions (or effects) until the requested outputs and post-conditions are resolved. They assume that service providers describe their services in their own service description language (USDL). Since all the service discovery and composition processes are performed in Prolog with Constraint Logic programming, services are pre-processed from USDL and transformed to Prolog terms. Pre-processing tends to be time consuming, which is sensitive in the case of runtime service composition support. Each time a service is added to the registry, the complete registry has to be pre-processed and updated with the new structure.

These dynamic service composition approaches bring automation and assistance to the service composition process, however they do not consider facilitating the participation of the end-user in the process. We have identified other approaches that tackle this problem and consider a more active participation of the end-user in the service composition process. Some of these approaches have been designed for specific application domains, such as, for example pervasive computing and context-aware applications. These approaches define mechanisms that take the user context, preferences and actions to drive the service composition process without explicitly informing the user about the use of this information.

Sirin et al. [19] propose a semi-automatic approach to support the composition of web services. This approach supports users in the discovery, selection and composition of semantic web services during each step of the composition process. The discovery process consists of finding matching services, which consist of web services that provide outputs that semantically match inputs of services that the user has previously selected for the composition. This corresponds to a *backwards-chaining service composition* process, as the user starts with the service that she wants, and then is guided through a process of discovering services to fulfill the preconditions of this service. After the discovery, selection and composition of services that meet the user requirements, the user validates the service composition, and the composition can be translated into an executable format.

Ben Mokhtar et al. [20] explore the use of semantic services as basic constructs for service delivery in pervasive computing environments. The different devices and functionality in the environment are exposed as services, which can be composed as required by the user, to deliver the functionality the user needs at a given moment. This approach differs from the approaches discussed before in the sense that it is limited to service delivery in specific pervasive computing environments, i.e., a specific and closed domain. This allows the approach to identify the different types of users of the environment and shape the supporting environment accordingly.

Sheng et al. [21] present a multi-agent based architecture to provide distributed, adaptive and context aware personalized services in wireless environments. In this approach, the end-user does not create a service composition, but service compositions are created beforehand and made available as template compositions that can be grounded (bound) at runtime to concrete services as the end-user selects them for execution. This grounding is done based on the user preferences and context.

Other related efforts [22-27] focus on the creation of intuitive visual mechanism to enable users to compose different services in so called *mashups*. We refer to [9] for a detailed discussion of related work in this area.

VII. Conclusions

This paper presents the A-DynamiCoS framework, which supports true user-centric service composition processes by allowing the different aspects of the service composition to be driven by the user. The service composition process is supported by a flexible orchestration of basic components that address the different phases of the service composition life-cycle (service discovery, service selection, service composition, service execution, etc.). In order to support this user-driven process, the framework has a front-end User Support component that gathers the user intentions at each step of the process. This component is developed by domain experts, who know the target user population to be supported in a given usage scenario. The front-end user support issues commands to the framework back-end system. Commands represent user intentions and are translated by the back-end

supporting system to supporting strategies that are realized by the basic components of the composition framework. This separation allows the front-end user support to be designed in many different ways and with different user interfaces, depending on the specific scenario, while the back-end supporting system remains generic.

To validate our user-centric service composition approach we have developed a prototype of the back-end supporting system, and then applied it to support two different use cases from two different application domains, namely e-government and entertainment. These domains have different requirements and types of users, i.e., they require different front-end user support mechanisms. The same back-end supporting system was used to support both use cases, in which the same set of primitive commands were issued in different orders and with different command flows. In this way we could show the applicability of our user-centric service composition approach to support users with different requirements and characteristics.

Future work includes the application of A-DynamiCoS to other use cases in order to evaluate the general applicability of the framework, and the development of mechanisms and tools to facilitate the definition of new primitive commands and to expose these commands to front-end User Support designers. We also intend to evaluate the usability of A-DynamiCoS, in order to measure the end-user perception of the service delivery process supported by the framework.

ACKNOWLEDGMENT

We thank Edwin Vlieg and Joni Hoppen dos Santos for their contribution to the improvement and evaluation of A-DynamiCoS through the implementation of the use cases.

REFERENCES

[1] S. Dustdar and M.P. Papazoglou, “Services and service composition - an Introduction,” IT - Information Technology, vol. 50, no. 2, 2008, pp. 86-92.

[2] E. Gonçalves da Silva, J.M. López, L. Ferreira Pires and M.v. Sinderen, “Defining and prototyping a life-cycle for dynamic service composition,” Proc. International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing (ACT4SOC), INSTICC Press, 2008, pp. 79-90.

[3] E. Gonçalves da Silva, L. Ferreira Pires and M.v. Sinderen, “Dynamic composition of services: why, where and how,” Proc. International Workshop on Enterprise Systems and Technology (I-WEI 2008), INSTICC Press, 2008, pp. 73-85.

[4] OASIS, “Web Services Business Process Execution Language Version 2.0,” 2007.

[5] E. Gonçalves da Silva, L. Ferreira Pires and M.v. Sinderen, “Towards runtime discovery, selection and composition of semantic services,” Comput. Commun., vol. 34, no. 2, 2011, pp. 159-168.

[6] E. Gonçalves da Silva, L. Ferreira Pires and M.v. Sinderen, “Supporting dynamic service composition at runtime based on end-user requirements,” Proc. International Workshop on User-Generated Services (UGS 2009), CEUR Workshop Proceedings, 2009.

[7] F. Lécué, E. Gonçalves da Silva and L. Ferreira Pires, “A framework for dynamic web services composition,” Proc. Emerging Web Services Technology, Volume II, Springer-Birkhauser, 2008, pp. 59-75.

[8] F. Lécué and A. Léger, “Semantic web service composition based on a closed world assumption,” Proc. Fourth European Conference on Web Services (ECOWS '06), IEEE Computer Society, 2006, pp. 233-242.

[9] E. Gonçalves da Silva, “User-centric service composition: towards personalised service composition and delivery,” PhD Thesis, University of Twente, Enschede, 2011.

[10] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design patterns, Addison-Wesley, 1995.

[11] R.T. Fielding, “Architectural styles and the design of network-based software architectures,” PhD Thesis, University of California, Irvine, 2000.

[12] W3C, “SOAP Version 1.2 Part 0: Primer (Second edition),” 2007.

[13] J. Hoppen dos Santos, “Public service improvement using runtime service composition strategies,” MSc Thesis, University of Twente, Enschede, 2010.

[14] E. Vlieg, “Usage patterns for user-centric service composition,” MSc Thesis, University of Twente, Enschede, 2010.

[15] J. Hoppen dos Santos, L. Ferreira Pires, E. Gonçalves da Silva and M.E. Iacob, “Public service improvement with user-centric service composition,” Proc. Ongoing Research and Projects of IFIP EGOV and ePart 2011, Trauner Verlag, 2011, pp. 276-284.

[16] R.M. Pessoa, E. Gonçalves da Silva, M.v. Sinderen, D. Quartel and L. Ferreira Pires, “Enterprise interoperability with SOA: a survey of service composition approaches,” Proc. International Workshop on Enterprise Interoperability, 2008, pp. 32-45.

[17] J. Pathak, S. Basu, R. Lutz and V. Honavar, “MoSCoE: a framework for modeling web service composition and execution,” Proc. IEEE 22nd International Conference on Data Engineering Ph.D. Workshop, IEEE CS Press, 2006.

[18] S. Kona, A. Bansal, G. Gupta and T.D. Hite, “Automatic composition of semantic web services,” Proc. International Conference on Web Services (ICWS'07), 2007, pp. 150-158.

[19] E. Sirin, J.A. Hendler and B. Parsia, “Semi-automatic composition of web services using semantic descriptions,” Proc. Workshop on Web Services: Modeling, Architecture and Infrastructure, 2003, pp. 17-24.

[20] S.B. Mokhtar, A. Kaul, N. Georgantas and V. Issarny, “Efficient semantic service discovery in pervasive computing environments,” Proc. ACM/IFIP/USENIX 2006 International Conference on Middleware, Springer-Verlag New York, Inc., 2006, pp. 240-259.

[21] Q.Z. Sheng, B. Benatallah and Z. Maamar, “User-centric services provisioning in wireless environments,” Commun. ACM, vol. 51, no. 11, 2008, pp. 130-135.

[22] J. Han, Y. Han, Y. Jin, J. Wang and J. Yu, “Personalized active service spaces for end-user service composition,” Proc. IEEE International Conference on Services Computing, 2006, pp. 198-205.

[23] Y. Han, H. Geng, H. Li, J. Xiong, G. Li, B. Holtkamp, R. Gartmann, R. Wagner and N. Weissenberg, “VINCA: a visual and personalized business-level composition language for chaining web-based services,” Proc. International Conference on Service-Oriented Computing, Springer-Verlag, 2003, pp. 165-177.

[24] V. Hoyer, K. Stanoesvka-Slabeva, T. Janner and C. Schroth, “Enterprise mashups: design principles towards the long tail of user needs,” Proc. IEEE International Conference on Services Computing, 2008, pp. 601-602.

[25] X. Liu, Y. Hui, W. Sun and H. Liang, “Towards service composition based on mashup,” Proc. IEEE Congress on Services, 2007, pp. 332-339.

[26] A. Ro, L.S.-Y. Xia, H.-Y. Paik and C.H. Chon, “Bill organiser portal: a case study on end-user composition,” Proc. International Workshops on Web Information Systems Engineering, Springer-Verlag, 2008, pp. 152-161.

[27] J.C. Yelmo, R. Trapero, J.M. Álamo, J. Sienel, M. Drewniok, I. Ordás and K. Mccallum, “User-driven service lifecycle management: adopting Internet paradigms in telecom services,” Proc. International Conference on Service-Oriented Computing, Springer-Verlag, 2007, pp. 342-352.