An Evolutionary Algorithm for Automatic Composition of Information-gathering Web Services in Mashups

Thomas Fischer, Fedor Bakalov, Birgitta König-Ries
Friedrich-Schiller University of Jena
07743 Jena, Germany
{fischer.thomas | fedor.bakalov | birgitta.koenig-ries}@uni-jena.de

Andreas Nauerz, Martin Welsch
IBM Research and Development
71032 Böblingen, Germany
{andreas.nauerz | martin.welsch}@de.ibm.com

Abstract—The idea behind mashups is to provide a mechanism that allows for more or less spontaneous combination of existing web applications. Users shall thus be enabled to combine data and services according to their needs. However, existing mashup frameworks require some programming knowledge, hence are not suitable for non-expert users. In this paper, we present a system that builds on existing Semantic Web research to achieve an automatic, ad-hoc generation of mashups thus eliminating the need for programmer involvement. At the core of our approach, there is an evolutionary algorithm that automatically composes different information web services based on semantic service descriptions. The information that has been retrieved from the invoked web services is automatically transformed into a semantic representation and presented as a mashup to the users of the system.

Keywords—Web Service Composition, Semantic Web Services, Evolutionary Algorithms

I. INTRODUCTION

Traditionally, when users have required new functionality or a new combination of existing functionality, they had to ask the IT department to write a suitable application. This was a slow and often bureaucratic process. Nowadays, many applications are offered over the Web and can be discovered and used by the end users without the need for help from the IT department. A number of such applications provide their functionality and information via so called Web Services, which allows users to combine them to fulfill the information needs they might have in a certain situation. Imagine, for instance, a financial manager reading about a planned merger of two companies. He might be interested to obtain additional information about a variety of aspects: How is the current trading price of stocks of the company? What are profit figures for the last quarter? How is the industry doing in general? What do we know about the people involved? Where is the company situated and which hotels are available?

It is quite likely that applications exist that can offer the required information. It is rather cumbersome, however, to manually find and combine the individual applications each time the user needs them. Furthermore, interests and context of the user are changing, thus, the set of required applications is not static, but needs to be adapted to the current situation. Mashups offer a programming model that allows for the easy, plug-and-play like combination of different web applications. An appropriate mashup framework would allow the user to use the chosen combination of tools automatically in the future. Unfortunately, the creation of mashups is not trivial. While by now many frameworks for mashup creation exist, all of them require some programming experience as we will discuss in Section II.

Based on this observation, we propose a method for the automatic generation of mashups. The basic idea (see Figure 1) is to use the information available about a core application and a user’s current task (which we assume can be found in a user model) to determine what additional information the user is interested in. We then employ an evolutionary algorithm to combine existing applications, to be more precise existing information gathering web services, in such a way that these information become available. The newly available pieces of information, which are classified in terms of an ontology, are then transformed into a mashup page to support the user. Figure 2 shows an simple example of the system. The initial text of a portlet has been extended by a map, which shows hotels on nearby locations. Furthermore, additional information for specific highlighted entities are available. The application automatically composed a company address service, a geocoding service and a service for finding hotels in nearby locations, because the specific user goal has been set to travelling. Thus, the map displays hotels of cities that are directly mentioned in the initial text as well as hotels of cities that are locations of companies of the text.

A brief overview of the overall system architecture which we used is shown in Section III, while a detailed description of the evolutionary algorithm can be found in Section IV. The proposed algorithm is a suitable solution...
to the problem as we will show with a thorough evaluation in Section V. We will show that the algorithm provides correct results and scales well - two equally important features in the mashup context. Finally, Section VI concludes the paper.

II. RELATED WORK

In this section we first present related work on mashups and web services to outline their inter-relations. We then give a brief overview on the huge work on semantic web service description and composition, because these semantic web technologies are used in our proposed approach to achieve an automatic generation of mashups. This overview serves two purposes: on the one hand, we need to introduce some foundations that our work builds on, on the other hand, we will explain where limitations of existing approaches are and why a new approach was needed.

A. Mashups and Web Services

1) Mashups: The research on mashups, which are a principle of the Web 2.0, has turned out a variety of mashup definitions [1, 2, 3, 4, 5, 6, 7, 8]. In general, the definitions provide a consensus over the integration and aggregation of different resources in mashups, while differences arise mostly from the types of considered resources. Often these resources are Web Services whereas the mashups themselves put a visual layer on top of these Web Services [9]. A comprehensive market overview of the different types of mashups and tools could be found in [9]. Furthermore, a huge set of example mashups is maintained by ProgrammableWeb\(^1\).

The approaches and tools for creating such mashups are very diverse. They range from fully manual development to fully automatic generation, which is proposed in this paper. An overview about the different approaches as well as there limitations has been already described in one of our previous papers [10], thus we give only a short abstract here.

Today, several tools are available for manual creation of mashups. Systems, such as Apatar\(^2\) [11], IBM Damia [12], JackBe Presto Wires\(^3\), Microsoft Popfly\(^4\), Yahoo Pipes\(^5\), Openkapow\(^6\), Proto Financial\(^7\), Anthracite\(^8\), Lotus Mashups\(^9\), Marmite [13], SABRE [14], provide users with a GUI environment where they can aggregate different data sources using certain operators and filters. They offer widgets associated with data sources (e.g. RSS feeds), which users can drag to the canvas and use to define how data should be aggregated. This manual connection is sometimes called wiring or piping of different modules, connectors, components or blocks. The available components provide different functionality (e.g. data retrieval, data transformation, data presentation etc.) and have to be connected to achieve the desired coordination of the mashups. The tools often support different data source types such as RESTful and / or SOAPful (e.g. Openkapow, Proto Financial, Anthracite) web services, database, spreadsheets and CSV files.

Furthermore, the creation of mashups is supported by various tools that are based on domain specific scripting languages such as Google Mashup Editor\(^10\) (GME), Web Mashup Scripting Language (WMSL) [15, p.1305], Dynamic Fusion of Web Data [4] [16], WSO2 Mashup Server\(^11\). In general, it seems to be too complicated for a non-developer, to create such scripts in an appropriate time, because more complex mashups will need a considerable amount of rather complex script code.

The present systems have a number of limitations. First, in order to create even a simple mashup, the user has to possess a certain level of technical knowledge. Second, mashups are mostly used to solve a short-term information need, thus they have to be created ad-hoc, for which users are not always willing to invest time manually assembling data. Third, the number of available data sources and aggregation operators are limited to what the system provides. Another important limitation is the lack of adaptivity. This means that if the data sources and functionalities of the provider change their structure or behavior the mashup application has to be reengineered by the endusers. It is therefore an important topic to achieve an automatic and adaptive creation of mashups.

Overall, the entirely dynamic and automatic generation of mashup applications has been only limitedly part of research activities. In [17], as one example, a mashup framework for automatic composite mashup applications

\(^1\)http://www.programmableweb.com

\(^2\)http://www.apatar.com/product.html

\(^3\)http://www.jackbe.com

\(^4\)http://www.popfly.com/

\(^5\)http://pipes.yahoo.com/pipes/

\(^6\)http://openkapow.com/

\(^7\)http://www.protosw.com/

\(^8\)http://www.metafy.com/products/anthracite

\(^9\)http://www-01.ibm.com/software/lotus/products/mashups/

\(^10\)http://code.google.com/gme/

\(^11\)http://ws02.org/projects/mashup
based on Lotus Expeditor is proposed. The framework focuses on mashups of non web service components (e.g. widgets), whereas we focus on the mash-up of background information provided by information web services. Furthermore, we have a focus on fast ad-hoc creation of personalised mashups, which requires the fast creation of user centric compositions of information Web Services even in large Web Service repositories.

2) Web Services: Web Services can be defined as functionality that could be engaged over the Web [18, p.520] and that make descriptions of their interfaces available. The Web Service Description Language (WSDL) provides the possibility to publish technical description of a web service [19, 20], but it does not allow to capture the semantics of the service, i.e., it describes interfaces, messages, and endpoints, but it does not define the meaning of the exchanged messages or the effects the service execution has. Web Services can be divided in SOAPful and RESTful services. While the name of the former is derived from the underlying Simple Object Access Protocol (SOAP), a protocol for exchanging XML messages, which can use, e.g., HTTP, SMTP as its underlying transport protocol [18, p.21-36][21], REST stands for Representational State Transfer)[22]. This means that the invocation of a URI responds a representation of a source that transfers the client application into a specific state. The HTTP response contains mostly a XML document that denotes the representation of this resource.

A subclass of web services that is of particular importance for this work, are information web services. These are web services that return information for a specific request and that achieve no real world effects (in contrast to, e.g. a book buy service).

B. Ontologies

Knowledge representation and ontologies are important for mashups, because mashups combine data from disparate sources that can be only combined in a more automatic way, if there is a shared understanding of the meaning of the data. The most prominent definition for ontology was given by Gruber [23], who specified an ontology as "an explicit specification of a conceptualization". Conceptualizations can be shared among agents. Therefore, the definition has been extended by Borst [24, p.12]: "Ontologies are defined as a formal specification of a shared conceptualization".

The Resource Description Framework (RDF) [25] is a framework for representing information on the Web. In general, RDF allows anyone to make statements about any resource, which could be a material or immaterial thing. A statement is defined as a triple, consisting subject s, predicate p and object o, written as p(s,o). This means that subject s has a predicate (or property) p with value o. RDF provides no means to define the terms of the vocabulary that is used throughout the statements. This is addressed by RDF Schema (RDF-S) [26] and the Web Ontology Language (OWL) [27]. RDF-S is a minimal ontological language. It has capabilities to define classes and properties, and enables the specification of how they should be used together.

The Web Ontology Language (OWL) is build on top of RDF and RDF-S. OWL provides the three sub languages OWL-Lite, OWL-DL and OWL-Full. The usage of a language depends on the needed expressiveness of the ontology. In our framework we use an OWL-DL ontology. OWL-DL is an expressive description logic (SHOIN(D)). The logical structure of a DL knowledge base is based on a so called TBox and a ABox (\( KB = (TBox, ABox) \)). The TBox contains intensional knowledge and is build through the definition of concepts and properties [28]. The ABox contains assertions about the named individuals in terms of the defined vocabulary. Furthermore, the ABox depends on the current circumstances and is part of constant change. A detailed overview about description logics can be found in [29].

C. Semantic Service Description

Service description languages are used to describe the functional and non-functional semantics of web services. Examples of such description languages are OWL-S [30], WSML [31], SAWSDL [32]. An overview about different semantic service description languages can be found in [33]. In the following we give a brief overview about OWL-S, which is used in our framework.

The Web Ontology Language for Web Services (OWL-S) is an upper ontology for web services that contains statements about the web service profile, web service model and web service grounding. The service profile is intended to support the selection of a web service and describes inputs, outputs, preconditions and results of a web service. The web service model describes how the web service could be used by a client. It could be used to make a more in-depth analysis for the selection of a web service or it could support the composition of web services. The OWL-S model of a web service is described by process descriptions. A process in OWL-S is defined by its inputs, outputs, preconditions and results. The preconditions and results can be described by a logic expression. Amongst others the specification supports KIF, PDDL, SWRL and since OWL-S 1.2 [34] also SPARQL [35] as logical languages. OWL-S defines atomic, simple and composite processes. An atomic process corresponds to an action that is performed in a single interaction. This means that there are no subprocesses and that the process could be directly invoked, if the preconditions are fulfilled. An atomic process is referenced to a grounding that specifies how to construct the messages.

D. Semantic Web Service Composition

It is likely that complex information needs can not always be served by a single web service, especially in the context of mashups. Therefore, it is important to be able to create a plan that represents a composition of different web services. Web service composition is a kind of a planning problem and involves search and logic inference of Artificial Intelligence (AI). Planning is the task of the creation
of a sequence of actions that achieve a desired goal [36, p.375]. This means that a planner has to create a sequence of web services that could be invoked by the software agent to achieve its present goal. Actions in AI planning based approaches are specified by preconditions and effects. The preconditions have to hold before the agent invokes the action. The effects describe how state changes after the execution of the action [[36, p.379]. In advance to the concrete planning process, the majority of planners transforms the goals, inputs and outputs of web services as well as preconditions and effects in a First-Order-Logic (FOL) problem representation. There are a multitude of approaches for Web Service Composition, which could be classified, in accordance to [37], as follows: static vs. dynamic composition, functional- vs. process-level composition. While in a dynamic composition approach the web services are planned at invocation time, the static composition first generates the plan and then invokes the web services. Functional-level composition means that a web service is considered as an atomic entity. This means that the web service is specified by its inputs, outputs, preconditions and effects (IOPE) and requires only a simple request-response interaction. Atomic web services are also those ones, which are provided as a black-box and thus the underlying behaviour and interactions are not visible. Instead, process-level composition considers the internal interactions of a web service [37], which would correspond to a more complex interaction schema.

The research on semantic web service composition states that there are still open topics on appropriate incorporation of user preferences in the matching process [38] [39, p.93-94] [40], scalability of planning algorithms [41, 37] as well as adaptive discovery and composition [37]. However, these limitations are important requirements in the mashup context. The proposed user model as well as the evolutionary web service composition algorithm address these open issues. For these reasons, we propose our own web service composition approach, which uses an evolutionary algorithm for state space traversal and a boot strapping evaluation function. While evolutionary algorithms has already been used in the web service challenge [42], we use a novelty evaluation function and we also reuse the semantic capabilities of OWL-S. In contrast to [42], we have a more detailed plan representation that allows parallel execution of web services in a composition. Furthermore, we also regard the invocation of the information gathering web services.

III. SYSTEM ARCHITECTURE

Our mashup framework provides users personalized mashups that are automatically generated based on the knowledge about the user and the information about the context of the acting user. The framework augments the documents that users are reading with background information and related content gathered from different sources and aggregated into one integrated tool.

In the case of the finance manager mentioned in the introduction, the framework, for instance, can display a side menu that shows the situation on the stock market, including the stock quotes of the companies mentioned in the text. Another side menu can be displayed to provide an executive summary of the technology mentioned in the article and the list of people from her department who are familiar with this technology.

Figure 3 illustrates the high-level components of the system architecture of our framework. In order to automatically augment the external or portal content (1) with relevant information, we need a component that is able to extract machine-readable semantics from the content (2). For this purpose, we use the Calais12 Web service and UIMA framework13. These analysis engines are able to extract certain types of entities, such as person, company, location, etc (3).

The semantics extracted from the content is used by the personalization engine (4) to identify the information-gathering actions that provide the user the information she may need in a given situation. Each action is described by its input and output concepts, which denote the input data and wanted information respectively. The selection of actions is based on the personalization rules defined in the personalization model and the knowledge about the user interests and expertise stored in the user model, both of which are based on the domain model. The domain model is based on an ontology. More information on all four components can be obtained from our previous publications ([43] and [44]).

After the personalization engine has identified the information-gathering actions relevant to the user, the engine invokes a web service composition request (5) for each of the selected actions. In the request, it specifies the semantic input data and the structure of information that must be obtained. Then the engine sends the request to the service composition module, the goal of which it is to find the services that provide the requested information. The composition module creates a composition (6) by a multi-objective evolutionary algorithm, which operates on the semantic web service descriptions provided by the application registry. The application registry maintains

Figure 3. System Architecture

12 http://www.opencalais.com/
13 http://incubator.apache.org/uima/
the references to the semantic web service descriptions, which are based on OWL-S [30], for each of the registered RESTful or SOAPful services. Furthermore, it maintains information about the inputs, outputs, preconditions and results of each semantic web service.

After the composition, the module provides the generated composition to the mashup handler (7), which in turn invokes the semantic web services (8,9) and requests the presentation module to display the mashup (10).

IV. EVOLUTIONARY ALGORITHM

In the remainder of this paper, we will concentrate on the service composition module. The module searches for a plan, or more precisely a web service composition, based on an evolutionary strategy, which traverses the search space through a special stochastic hill-climbing. This means that the algorithm evaluates and maintains simultaneously a population of states (plans) of the search space. New states (plans) are created from old states by special operators for mutation and recombination, which are based on the principles of the natural evolution. The proposed planner is a functional-level static planner that considers atomic web services that are based on a simple-request response interaction schema. Dynamic planning with the proposed evolutionary process seems to fit not very well, because this would in general cause the invocation of a lot of unnecessary web services. Furthermore, the invocation of the web services would slow down the whole search. The evaluation function therefore relies on the specified semantic web service descriptions. It is necessary to have an expressive web service description to specify the correctness and completeness of such plans [45].

Section IV-A describes the concepts of the evolutionary process in the context of this paper. Section IV-B defines the formal representation of the plans. Section IV-C deals with the definition of the final search problem for web service composition planning.

A. Evolutionary Process

The algorithm starts with a random set of possible solution candidates (web service compositions, plans). Based on the calculated performances of the candidates the parent selection calculates which individuals are admitted for recombination. The selection is based on a tournament between the individuals. The recombination is an important evolutionary factor that is based on a two-point crossover that pairwise recombines the interconnection of parts of two candidate plans. This step can lead to very different phenotypical characteristics. The newly generated child plans are then mutated in different ways with a predefined probability. The mutation changes the web services identifiers of the plan as well as their ordering. In addition, mutation can grow and shrink plans, because the final plan length is not known in advance. Mutation and recombination lead to plans that are often not compliant with specific assumptions, which are described in Section IV-B. Therefore, the evolutionary process contains an operator that repairs the plans. The repaired child plans are then weighted by the evaluation function and added to the existing population. However, the plan population has only a predefined size, which requires to perform a so called environment selection that determines which plans survive for the next generation of the population. The algorithm has two conditions that are described in detail in Section IV-C.

B. Plan Representation

The problem of web service composition planning is relaxed to the search for an appropriate Directed Acyclic Graph (DAG) (see phenotype of Figure 5). The Graph \( G = (V,E) \) is based on a set of nodes, whereby it is assumed that each node represents an atomic web service. Each web service is denoted by its identifier defined by the application registry. Furthermore, it is assumed that each web service can be used only once in the plan. An arrow from node \( A \) to node \( B \) denotes that the web service of node \( A \) should be invoked before the web service of node \( B \). This is important, because this enables the web service of node \( B \) to utilize the information provided by the former web service.

The DAG represents the phenotype of the plan representation. However, the evolutionary operators for recombination and mutation operate on the formal genotypical representation. A DAG could be represented in different ways, such as in a standard matrix or adjacency-list encoding [46, p.22].
an adjacency-list representation (Figure 5). The genotype \( G \) is a tuple \( G = (G.R, G.F) \), whereby \( G.R \in \mathcal{G} \) denotes the representation of the DAG and \( G.F \in \mathbb{R} \) denotes the value of the evaluation function of the plan. Each genotype \( G.R \) contains a list \( K \) of web service identifiers and a second list \( T \) that contains the topological information of the composition, to be more specific a list of sets of topological levels. A topological level is equal to a position (index) of a service in the list \( K \). In Figure 5 the topological levels are denoted by “L”, however, this should only outline the principle. In the concrete representation the level is simply an integer value. Therefore, a topological level \( l \in T[i] \) corresponds to a directed arc \((K[i],K[l]) \in E\). For instance, \( L4 \in T[2] \) denotes that there is an arc \((K[2],K[4])\). This means that the outputs of web service 1 are used in the inputs of web service 8.

This encoding also enables the specification of a lightweight rule that determines the existence of cycles and allows an efficient repair of the genotypes. An encoded plan of length \( L \) has no cycles, if \( \forall l \in T[i] : l > i \ i = 0, 1, 2, \ldots , L \) holds true.

Instead of a direct evaluation of the phenotypical plan representation by the function \( f \), the weighting of a plan is based on the genotypical evaluation function \( f_{WC} \). Furthermore, the decoder function \( decode \) determines the appearance of a plan based on the genotypical data. The weighting value of each plan is then used in the selection step. Afterwards, the evolutionary operators for recombination and mutation modify the genotype information.

C. Formal Problem Definition

This section defines the final search problem for web service composition planning, which is represented as an optimization problem of an objective function. In the following a specific solution state (plan, genotype) in the state space is denoted by \( x \) to simplify the notation of the problem.\(^{14}\)

A single-objective optimization problem \((\Omega, f, >)\) is defined by the state space \( \Omega \), a weighting function \( f : \Omega \to \mathbb{R} \) as well as a relation for comparison \( > \). The weighting function assigns a rating to each solution candidate (state) \( x \in \Omega \). The set of global optima is defined by \( \chi = \{ x \in \Omega \mid \forall x' \in \Omega : f(x) \succeq f(x') \} \) (\cite[p. 21]{47}).

However, web service composition planning is based on multiple objectives such as correctness, completeness, length, reliability, price etc. of plans. These objectives have to be considered for the planning process, which makes the search for a solution much more difficult than in single-objective optimization problems. While single-objective problems may only have one unique optimal solution, a multi-objective problem could have a set of possible solutions vectors. The different objectives are computed by objective functions. Since the objectives are conflicting (e.g. completeness versus length) as well as operate on different scales (e.g. price versus performance), they have to be aligned appropriately.

The general multi-objective problem searches for a solution \( x^* \) (a state of the state space) that optimizes the vector function \( \vec{f}(x) = [f_1(x), f_2(x), \ldots, f_k(x)]^T \). It is clear that in most cases no optimum exists such that \( \forall x \in \Omega : f_i(x^*) \geq f_i(x) \) holds true. This means that there is not a single solution \( x^* \) that strictly dominates all other solutions.

Multi objective problems are denoted by a so called Pareto-optimum, which represents the set of possible solutions that are efficient. Thus, this set contains only solutions that are not dominated by an other solution. The selection of a suitable solution out of the Pareto-optimal set depends on the preference structure of the specific agent and can be handled a priori, a posteriori or progressive to the optimization. Bleul et al. \cite{42} state that the Pareto-optimization has disadvantages for evolutionary web service composition planning, because it spreads the search over the complete Pareto frontier and slows down the performance. In fact, in the context of the proposed framework it is appropriate to specify the preferences a priori, because the preference information are available from the specified user model.

The final web service composition optimization problem is defined as \((\zeta, f_{WC}, >)\). \( f_{WC}(x) = w_1 \cdot completeness(x) + w_2 \cdot correctness(x) + w_3 \cdot length(x) \). We will discuss the individual factors in more detail below.

The search space \( \zeta \) represents all possible permutations of web services. Furthermore, the evaluation function \( f_{WC} \) is assumed to be maximized for \( x \in \zeta \). The main objectives correctness, completeness and length are weighted by a specific linear combination. It is important to note that this handling of goal conflicts is only possible for such basic and general objectives. If other more user-sensitive preferences (e.g. reliability, security, price, performance, integrity, accessibility, availability, compliance \cite{18}) should be incorporated into the evaluation function, alternative decision theoretic approaches have to be utilized.

The function \( f_{WC} \) has to provide a gradual weighting \cite[p. 23]{47} to avoid an uninformed search. Thus, the objective functions should provide a gradual weighting too. For instance an absolute weighting of the correctness would only determine if a possible web service composition is correct or not correct. This would lead to a lack of information about partial good solutions, and thus the direction of the search could not be specified as detailed as in a gradual weighting. In the following we explain the calculation of the different objective functions of the evaluation function.

1) Calculation of the Objective Completeness: The completeness measures to which degree the goal of the planning process is achieved. The goal representation is based on a list of desired output resources. Algorithm 1 gathers the output resources of all web services of the genotype and merges them into one comprehensive set. The operation "containsAll" checks, if for all desired

\(^{14}\)Each solution refers to two lists that represent the web services and the topological information.
mashup output resources \( w \in OUT_M \) there is a resources \( c \in OUT_C \) such that \( c \) subsumes \( w \) (\( c \sqsubseteq w \)). This means that it checks that each output resource of the desired mashup is addressed by an either equal or more specific output resource of the composition.

**Algorithm 1 Completeness of composition**

Require: \( KB = (TBox, ABox) \)

Require: \( OUT_M \) {desired mashup output resources}

Require: \( G \) {genotype of the composition}

\( WS_i \) {web service in the registry} \( c \) {composition completeness \( \in [0, 1] \)}

\( OUT_C \) {composition output list}

for all \( WS_i \in G.R \) do

\( OUT_C \leftarrow OUT_C \cup WS_i outputs \)

end for

\( c \leftarrow containsAll(OUT_C, OUT_M) \)

return \( c \)

However, there are often web services that do not directly contribute to the goal state, but instead contribute to the overall plan. This means that intermediate web services have to be considered, by an additional heuristic that counts the *number of promising web services* (Algorithm 2). A promising web services is a web service which either has an output that is an desired mashup output resource or that has an output that is required as an input of another promising web service. Both heuristics drive the optimization towards the objective of completeness.

**Algorithm 2 Number of promising services**

Require: \( G \) {genotype of the composition}

Require: \( OUT_M \) {desired mashup output resources}

\( PWS_i \) {empty set for the promising web services}

\( WS_i \) {web service in the registry}

\( k \leftarrow 0 \)

for \( k \leq threshold \) do

for all \( WS_i \in G.R \) do

if \( WS_i \) has an output resources that subsumes a resource of \( OUT_M \) then

\( WS_i \cup PWS \)

\( WS_i inputs \cup OUT_M \)

end if

end for

end for

return \( PWS.size \)

2) *Calculation of the Objective Correctness*: The heuristic correctness function (Algorithm 3) utilizes the input and output information of the OWL-S web service descriptions. It checks for each web service, if the inputs are served by resources from the mashup content or by an output resource of one or more predecessor web services. For the specification of the predecessor web services the algorithm utilizes the topological information of the genotype.

3) *Calculation of the Objective Length*: The length of a plan is an important objective that should be *minimized*, because it is assumed that unnecessarily long plans lead to higher execution times and unnecessary costs. The calculation of the length is denoted by the count of web services in the genotypical encoding. However, tests of the algorithm have shown that the algorithm performs better, when the count of unimportant web services is considered, which could be calculated by:

\[ count \text{OfUnimportantWebServices} = length - count\text{OfImportantWebServices}. \]

4) *Composition Simulation*: The heuristics are not able to prove exactly if a composition is definitely executable. Therefore, we have developed a simulation of the plans. The whole simulation starts with a temporal OWL ABox which is represented in RDF. This ABox contains the semantic information from the initially input (portal page) of the framework, which is part of the composition request. The simulation is performed for all web services of a found candidate solution, whereby it uses the topological information of the solution candidate. For each web service the simulation checks the precondition by an SPARQL [35] ASK query. If the precondition is true, then a SPARQL CONSTRUCT query is executed and the resulting simulated statements are added to the temporal ABox as new available information pieces. These new information are then considered for the subsequent semantic web services. The simulation is successful if the preconditions of all web service could be served. In

![Algorithm 3 Correctness of composition](algorithm3)

Image 310x126 to 539x201

![Figure 6. Simulation of a semantic web service versus real invocation](figure6)

Figure 6. Simulation of a semantic web service versus real invocation

our simulation, the preconditions describe the structure of required information and the effects describe the structure
of generated output statements for each web service. Each precondition is described by a SPARQL $\text{ASK}$ query, which evaluates if the structure of the required input statements could be served by the available $\text{ABox}$ RDF statements. If this is true, an SPARQL $\text{CONSTRUCT}$ query is used to generate the structure of triples that are created by the invocation of the web service (see also Figure 6). In the simulation the generated triples contain only the types of instances as well as blank nodes that are used to create references between different instances. This approach is similar to a real execution, with the difference, that this is much faster than a real invocation and that it creates no unnecessary costs. Basically, triples are required as inputs of the semantic web service and output triples are generated by the web service. In a real execution, this is achieved through lifting and lowering schema mappings, which are well known from SAWSDL [32]. The following listings provide an example, whereby the prefixes in the SPARQL queries have been omitted.

### Listing 1. SPARQL-based Information-Precondition

```xml
<process:hasCondition>
<expr:SPARQL-Condition>
ASK { ?y rdf:type minerva:Stock .
?y psys:mainLabel ?name .
}
</expr:SPARQL-Condition>
</process:hasCondition>
```

The example is based on a financial stock quote web service described by OWL-S. The stock quote service requires input instances of the type “minerva:Stock” that are specified by their “mainLabel”. The SPARQL $\text{ASK}$ query (see Listing 1) is used throughout the planning to evaluate if this precondition is fulfilled for a given RDF representation of the current $\text{ABox}$. Listing 2 specifies the result of the stock quote web service. If the precondition is fulfilled, then the SPARQL $\text{CONSTRUCT}$ query is used to generate the output information. In this case, the stock quote web service creates a “minerva:hasQuote” object property for the existing stock instance. The object property relates a stock quote instance that is denoted by a blank node. The blank node specifies that it is of type “minerva:StockQuote” and that it is described by data and price. It is important to note that in the simulation only one instance of “minerva:StockQuote” is created, instead in the real world invocation the service can respond a set of stock quotes. However, only the structure of available information is important to determine the correctness of the composition during the simulation.

5) **Stop Condition:** The SPARQL queries lead to a timely costly simulation. Therefore, it is not appropriate to simulate each plan in the evolutionary algorithm. The heuristics are therefore used to drive the optimization towards a promising solution candidate (plan, state), which is then evaluated by the more exact simulation (see also Figure 4). A promising solution state is a plan that is heuristically complete and correct (see calculation above). A final solution is found, if this plan could be simulated such that all web service preconditions could be served. The algorithm creates no plan, if the algorithm exceeds a given number of iterations (generations). In this case the original content (no mashup) is displayed to the user.

The previous sections has explained the evaluation function. More details on the evolutionary algorithm, the evolutionary operators as well as the calculation of the objective function can be found in [48].

### V. EVALUATION

The algorithm has been part of a quantitative evaluation to test the scalability and correctness of this approach.\(^{15}\) An evaluation of the algorithm requires an appropriate test collection. In the context of this paper, the test collection should be based on services that are described with OWL-S service descriptions. Furthermore, the preconditions and results of the OWL-S service descriptions should be based on SPARQL (as explained in Section IV-C). In addition, the underlying web services should be atomic and invokable in reality. Furthermore, we need a large test collection to test the scalability of the approach.

Unfortunately, there is no standard test collection for web service composition that meets our requirements. This is a general problem also mentioned in other papers [49, 50, 37]. Therefore, we were required to adapt existing test collections. We have extended a test collection of 1000 OWL-S web service descriptions that has been retrieved from the Online Portal for Semantic Services (OPOSSum) [51] and extended them by additional web service descriptions. This has been done to increase the size of the search space, which is important to show the scalability of the algorithm. The planning is based on the abstract semantic descriptions of the web services, however, to test the automatic invocation of the web services, some descriptions refer to technical groundings that allow an access to corresponding real world web services.

In this paper, we consider a planning problem that requires the composition of five financial information web services and which also involves the search for intermediate web services that do not directly contribute to the goal information state. We concentrate on the analysis of the performance and scalability of the algorithm. For the given problem the planner has been randomly executed.

\(^{15}\)The evaluation has been performed on an Intel(R) Core 2 CPU 6400 with 2.13Ghz and 2GB RAM. The main memory has been restricted to a maximum of 1024 MB.
fifty times to evaluate the performance.\textsuperscript{16} Figure 7 shows

![Figure 7. Performance Analysis](image)

the count of runs of the algorithm with respect to the count
of generations until it has found a solution for the problem.
The count of runs has been aggregated into classes of 1000
generations. For two runs the algorithm has been stopped
at the maximum of 10000 generations without a solution
( class(9000,10000] ). The other runs solved the problem
on the average in 25 seconds. It can be concluded that the
algorithm is able to solve such problems and that they can
be solved in an acceptable time regarding a web service
registry of 1000 web services. 80\% of the runs require
between 1000 and 5000 generations to come to a solution.
Furthermore, there are only a few runs that require much
more or much less time than the corpus of the runs.

The \textit{scalability} in dependence to the registry size is also
very important. Therefore the planner has been executed
iteratively with registry sizes between 150 and 1000 reg-
istered web service descriptions. For each registry size
the planner has been run for 9 different seed values of
the random number generator to ensure the comparability.
Figure 8 shows the mean of the \textit{count of evaluated states}
in dependence to the registry size. The count of evaluated
states is calculated by the number of invocations of the
evaluation function until a solution has been found. For
instance, the evaluation function could evaluate 1000 states
(plans) of the state space and then have a correct and
complete solution. In this case, the measure is 1000. In
all runs of this test the algorithm has found a correct and
complete solution.

It can be concluded, that an increasing registry size
requires the evaluation of more states. This was expected,
however, it is important to note that the state space
increases much faster than the number of states the algorithm
has to evaluate. For instance, for 100 web services, the
state space has approximately 100! ~ 9.3E157 possible
states, whereas for 1000 web services the state space has
1000! ~ 4.0E2567 possible states. At the same time, the
mean of required states has only multiplied by about five.

\textsuperscript{16}The parameters of the algorithm have been the following: Library
Size: 1000, Population Size: 8, Selection Size: 8, Selection Type: Tournament Selection, Crossover Type: Two-Point Crossover, Service
Mutation Probability: 0.2, Topology Mutation Probability: 0.7, Grow
Probability: 0.3, Shrink Probability: 0.3.

However, one can see that the standard deviation of the
count of evaluated states increases, if the registry size
contains more web services. This is in accordance with
the previous result, where we have concluded that only a
few runs perform really bad or really good. However, apart
from the outliers, the algorithm has a good scalability even
for large registry sizes.

In this paper we do not compare these results with
other frameworks for web service composition, because
this would be only appropriate if all evaluations would rely
on a standard test collection of real world SOAPful and
RESTful Web Services and the same composition requests.
This is in accordance with Küster and König-Ries \cite{52},
who stated that the past investments in test collections
does not reflect the huge research activities in semantic
web services. The evaluation of the algorithm based on a
huge standard test collection of web services that could
be also invoked in reality is thus part of important future
work.

\textbf{VI. CONCLUSION}

Mashups are an attractive means to allow for sponta-
neous combination of applications to provide new func-
tionality. In contrast to existing mashup creation frame-
works, which rely on some programming knowledge by
the user, we propose the automatic generation of mashups
via an evolutionary algorithm. Based on information about
the desired functionality, this algorithm automatically de-
termines an appropriate combination of existing web ap-
plications. We proposed a detailed plan representation and
a mean to simulate a found web service composition with
SPARQL. We have thoroughly evaluated the algorithm and
shown that it provides correct results within a reasonable
time, even if the number of candidate applications is high.
Thus this work addresses the open topic of scalability
of web service composition. In our ongoing work, we
investigate, how the determination of required function-
ality, which is based on a rather simple user model up to
now, can be improved and personalized by an ontology-
based user and task model. This will improve the adaptive
behaviour and leads to research on user-centric ad-hoc
composition of web applications.