Dynamic Composition of REST services

Jesus Bellido

Abstract—Service composition is one of the principles of service-oriented architecture; it enables reuse and allows developers to combine existing services to create new services. Dynamic composition requires that service components are chosen from a set of services with equal or similar functionality at runtime. The adoption of the REST services in the industry has led to a growing number of services of this type, many with similar functionality. The existing dynamic composition techniques are method-oriented whereas REST is resource-oriented, and considers only traditional services. The REST architectural style has attracted a lot of interest from the industry due to the non-functional properties it contributes to Web-based solutions. In this thesis, we contribute to the area of web service composition in REST by proposing three techniques oriented to improve static and dynamic composition of this type of service. First we introduce a technique for static composition proposing a set of fundamental control flow patterns in the context of decentralized compositions of REST services. In contrast to current approaches, our proposal is implemented using the HTTP protocol and takes into account REST architectural principles. Afterwards, we present a technique to improve the dynamic composition in security domain extending ReLL to ReLL-S and allowing a machine-client to interact with secured resources, where security conditions may change dynamically. Finally, we propose SAW-Q, an extension of Simple Additive Weighting (SAW), as a novel dynamic composition technique that follows the principles of the REST style. SAW-Q models quality attributes, in terms of response time, availability and throughput, as a function of the actual service demand instead of the traditional constant values. Our results validate our main hypotheses indicating improvements with respect to alternative state-of-the-art methods. This also shows that the ideas presented in this thesis represent a relevant contribution to the state-of-the-art of REST service compositions.

Keywords—SOA, Web Services, Dynamic Compostion, REST, scalable architectures

1 INTRODUCTION

Web services are software entities that provide a determined functionality, they are platform-independent and can be described, published and consumed following a defined set of standards such as WSDL, UDDI and SOAP. These characteristics emphasize loose coupling between components and allow designers to develop interoperable, and evolving applications that can be massively distributed. Web services also promote the reuse of software, which reduces development costs, increases maintainability, and enables organizations to create composed services by combining basic and other composed services resulting in new software with aggregated value [1].

Service composition is performed statically if it occurs at design-time or dynamically if it occurs at runtime. Also, depending on how the composition is made, it can be manual or automatic. The dynamic and automatic service composition is an active research area due to the benefits of reduced need for pre-installation and configuration of the service composition, adaptability to change and contexts, high decoupling and personalization, smaller development time, and reuse [1]. However, static and manual techniques dominate in the industry due to the complexity of discovering services dynamically (WSDL descriptions lack domain semantics) and automatic composition. In addition, virtually all research on service composition focuses on the classic Web services (i.e. based on the WSDL, UDDI and SOAP standards). Currently, emerging services technology such as REST [1] is becoming an alternative to conventional services in the industry.

Unlike classical Web services which are centralized and operation-centric, the REST architectural style is resource-centric (e.g. `www.puc.cl/course/12`) that are manipulated through a limited set of standard and known operations (e.g.`HTTP`. `GET`, `POST`, `PUT`, `DELETE`). REST services are popular because they allow

• *J. Bellido is with the Department of Computer Science, Pontifical University, Chile.*
E-mail: jbellido@uc.cl

Manuscript received April 19, 2005; revised January 11, 2007.

1. Representational State Transfer.

massive scalability, decoupling, and fault tolerance (e.g. Amazon API, Facebook API, etc.). REST has attracted the interest of the scientific community to serve as a supporting framework for defining business processes and service composition. A REST resource should be able to be discovered by a machine at runtime, and it should be able to understand the mechanism of interaction with the resource (e.g. to use `GET` to read, and to use `DELETE` to delete it). This property would make possible to determine at runtime (dynamically) if the resource serves the consumer purposes, as well as to determine the order in which the operations, that change the resource state, should be invoked. Thus, it would be possible to generate workflows that implement B2B processes dynamically and automatically.

Currently, this is not possible because the REST resources lack a description that encapsulate domain semantics and can be interpreted by a machine. Moreover, current techniques of service composition are oriented to classic Web services that invoke an unlimited set of operations, rather than resources whose state are transformed by invoking a limited set of operations. The overall objective of this thesis is to contribute to the state of the art research on static and dynamic composition of REST services through the design and implementation of a dynamic composition model. In particular, we propose a RESTful decentralized, stateless composition model, a QoS model focused on security in order to determine the feasibility of service composition at runtime, and a QoS model focused on scalability, throughput and response time in order to dynamically select the best service component automatically.

2 BACKGROUND

2.1 Service Oriented Architecture

Service orientation has existed for some time and has been used in various contexts with different purposes. The most used form of this term has been to identify approaches that focus on the separation of concerns, which means that the logic required to solve a large problem that can be developed and managed, if such logic is decomposed into a collection of related pieces, and each piece gives a solution to a specific part of the problem [2].

This approach transcends technology but when combined with software architecture, service orientation acquires a technical connotation. Service-oriented architecture (SOA) is a model in which an application's logic is decomposed into several smaller units that together implement a larger piece of business logic. SOA promotes that these individual units exist independently but not isolated from each other, in fact they could be distributed. This requires that these units operate on the basis of certain principles that allow them to evolve independently while maintaining uniformity and standardization among them. In SOA these logical units are known as services.

Web services encapsulate business logic and can provide solutions to problems of various sizes. The service logic may include the logic provided by other services, in this case, the service is called a *composed* service, and the logic providers are called *component* services. A component service may be responsible for a single or a complex task. Web services interoperate among them due to a common understanding provided by services' descriptions. The description of a service determines the service endpoint, the data received and the expected data returned by the service through message passing. In this way, programs or services use a description and a message channel independent from a protocol to interact and create a loosely coupled relationship.

In summary, SOA services maintain a relationship that minimizes dependencies (loose coupling), adhere to communication agreements (service contract), independently manage the logic they encapsulate (autonomy), hide logic that is not relevant to a determined context (abstraction), separate responsibilities in order to promote reuse (reusability), can be coordinated and assembled to form new services (composability), avoid to retain or store information specific to an activity (interaction stateless), and are designed to be described so they can be found and executed through discovery mechanisms (discoverability). The collection of services raises an inventory of services that can be managed independently [1].

SOA establishes an architectural model that aims to improve the efficiency, agility and productivity when developing software. It places services as first-class entities through which the business logics are implemented in alignment with service-oriented computing goals.

2.2 Web services

According to the context in which they are used, Web services may assume different roles: providers, clients and intermediaries. A Web service plays a provider role if it is invoked from an external source, and a client role if it invokes a provider service. Client services look for and evaluate which is the appropriate service to invoke based on the provider service description. Intermediary services are those which play a role of routing and processing messages sent between client and provider services until they reach their destination [1].

Traditional Web Services

The platform to implement Web services has been traditionally defined by a set of industry standards. This platform is divided into two generations, each associated with a set of standards and specifications [1]. The first Web services generation is comprised of the following technologies and specifications: Web Service Description Language (WSDL), XML Schema Definition Language (XSD), Simple Object Access Protocol (SOAP) and Universal Description Discovery and Integration (UDDI). These specifications have been widely adopted in the industry; however, they lack information about service quality, which is required to address mission critical functionality at production level. The second generation adds extensions (WS-* extensions) to fill the gap, related to service quality, left by the first generation. The main issues addressed by these extensions are service security, transactions, reliable messaging services, among others.

A traditional Web service is comprised of [1]:

- A service contract that is, a WSDL document describing the service interface (endpoint, operations and parameters) and an XML schema definition defining data types.
- The program logic. This logic may be implemented by the service itself, or inherited and wrapped by the service so that the legacy functionality can be consumed.
- Message processing logic consists of a set of parsers, processors and service agents. The runtime environment generally provides this logic.
- Software components that implement non-functional requirements defined by the WS-* standards extension.

REST Services

REpresentational State Transfer (REST) [3], [4] is an architectural style that underlies the Web. This approach is another way of implementing a service that follows the design constraints and requirements specified by this architectural style. Each constraint can have positive and negative impact on various non-functional attributes. The non-functional goals of REST are: performance, scalability, simplicity, modifiability, visibility, portability and reliability. An architecture that lacks or eliminates one of the REST constraints is usually not considered a REST architecture.

The REST architectural style constraints are:

- Client-Server: Enforces separation of responsibilities between two components of the architecture, the client and the server. This establishes a distributed architecture where each component evolves independently. This restriction requires the server to process the requests sent by the client.
- Stateless: Determines that the past communication between the client and the service are not kept stored on the server-side (service), that is, the interaction state is not remembered by the service. This implies that each client request must contain all the information necessary for the service to process the request and respond accordingly, without the use of session information.
- Cache: Services responses may be temporarily stored in a Web intermediary component (e.g. routers, gateways, proxies, etc.) and thus avoid the service to process a similar request again, diminishing the workload on the service-side.
- Uniform Interface: This constraint states that the architectural components must share a single interface to communicate, which is detailed below.
- Layered System. A REST-based solution can contain multiple architectural layers. These layers may be added, modified and rearranged according to the evolvability need of the solution.
- Code On Demand: This is an *optional* restriction that allows that some logic is dispatched from the server to the client, to be executed on the client-side. It allows to customize Web applications

The REST uniform interface is a set of architectural constraints that differentiates REST from any other style:

- Resources must be uniquely identified (e.g. through a URI), and the identifiers must be opaque to prevent coupling, that is, the structure of a URI shall not include any particular meaning that can be guessed by a customer.
- Resources' state must be manipulated through operations defined by standard (or ad-hoc) network protocols. For example, for the case of HTTP, the operations are `POST` that initialize the state of a resource whose identifier is unknown and could possibly create a subordinate resource, `GET` which obtains a representation including the current state of a resource, `PUT` that modifies the state of an existing resource, `DELETE` that indicates a request to eliminate a resource (but could be ignored by the server). `GET`, `PUT` and `DELETE` operations are idempotent and reliable. The `POST` operation allows an unsafe interaction since the invocation of this operation may cause changes to the server [3].
- A resource can support multiple representations that encode the state of the resource in a particular format (e.g. XML denoted by `application + xml`). The format can be negotiated through HTTP headers to facilitate interoperability between client and server.
- The HATEOAS (Hypermedia as the Engine of Application State) property indicates that the state transitions modeled in a Web application (e.g. buy a book, pay the bill, provide the deliver address, etc.) are served as hyperlinks that indicate the user the set of actions available to him or her at a given time (representation).

2.3 Service Composition

Software integration involves connecting two or more applications that may or may not be compatible, even when they are not built on the same platform or were not designed to interact with one another. The growing need for reliable data exchange is one of the strategic objectives of integration. Web services are inherently designed to be interoperable and are built with the knowledge that they will have to interact with a long-range of potential service consumers.

Web service composition is a technique that coordinates the combination of service components operating within an agnostic business process context. A composed service is comprised of component services that have been assembled in order to provide the functionality required to automate a task or a specific business process. These service components may be part of other compositions. The ability of a service to be naturally and repeatedly composable is essential to achieve the strategic goals of service-oriented computing.

When Web services are composed, the business logic of the composed service is implemented by several services, which allows the creation of complex applications by progressively adding components.

Service composition is associated with business processes automation. When a business process workflow is defined, several decision points are created in order to define the dataflow and actions according to variables and conditions evaluated at runtime. A business process definition can be done manually if a person defines the sequence of invocations, or automatically if an algorithm defines such sequence. In addition, service components can be chosen at design time (static composition) or at runtime (dynamic composition).

Orchestration and Choreography

In service composition, the coordination of service components invocation can follow two styles: orchestration or choreography. The orchestration is a centralized control of a business processes execution; it defines the business logic and the order of service invocation and execution. Unlike the orchestration, the choreography has a decentralized approach; services collaborate and determine its role in the interaction.

Orchestration is the process by which various resources can be coordinated to bring out the logic of a business process. Orchestration technology is commonly associated with a centralized platform for activities management [5]. Service orchestration encapsulates a service business process and other services invocations. The most widely used technology in the industry to implement orchestrations is the Web Service Business Process Execution Language (WS-BPEL) [6].

REST's navigational style naturally supports a workflow style of interaction between components.

However, interaction is decentralized, components are loosely coupled and can mutate at any time. This characteristic, called evolvability, poses a challenge to service composition since components (resources) may change unexpectedly. Hence, clients must make few assumptions about resources and must delay the binding with the actual resources up to the invocation-time (dynamic late binding) [7].

REST composition research focuses on orchestration, with JOpera [7] being the most mature framework. In JOpera, control and data flow are visually modeled while an engine executes the resulting composed service. In [8], control and data flow is modeled and implemented using a Petri Net whereas interaction and communication with the resources themselves is mediated by a service description called ReLL [9]. In [10], control flow is specified in SPARQL and invoked services could be WSDL/SOAP-based endpoints or RESTful services (i.e., resources); from the orchestrator perspective, services are described as graph patterns. In [11] resource's graph descriptions are publicly available (can be discovered using HTTP OPTIONS). A set of constraints regulates when certain controls can be executed on resources (e.g., a required state), so that an orchestrator engine could perform a composition, but no indication is given about how to express such constraints. The two former approaches support dynamic late binding and the hypermedia constraint. An RDF-based approach for describing RESTful services where descriptions themselves are hyperlinked is proposed in [12]. The approach is promising for service discovery at a high level of abstraction; however, no support for dynamic late binding is provided and the composition strategy is not detailed.

Choreographies can be described from a global and local (one party) perspective. WS-CDL [13] is a W3C candidate recommendation that describes global collaboration between participants in terms of types of roles, relationships, and channels. WS-CDL defines *reactive* rules, used by each participant to compute the state of the choreography and determine which message exchange will or can happen next. Stakeholders in a choreography need a global picture of the service interaction, but none of them sees all the messages being exchanged even though such interaction has an impact on related parties [14].

In [15], REST-based service choreography standards evolution is presented. Business processes, modeled as choreographies, are implemented as single resources at a higher level. The lower level comprises the resources themselves (process instances). Although the approach provides process state visibility, it is not clear whether the higher level corresponds to a centralized orchestration or to a partial view of choreography.

2.4 REST services composition

Resources and resource collections are the components in a RESTful scenario [7], [16]. Unlike WSDL-based services, REST resources have standardized, few, and well-known assumptions at the application level (i.e. the Web) instead of the domain level. Resources must be identified with a URI (Universal Resource Identifier) that binds the proper semantics (at the application level) to the resource ([3] section 6.2.4), and must be manipulated through *links* and controls (i.e. an HTML form) embedded in a resource's *representations* (e.g., HTML page). Representations dynamically determine the set of resource's state transitions available to consumers (Hypermedia as the engine of application state [3]).

Composition requirements that are specific to REST are *dynamic late binding*, that is, the URI of the resource to be consumed can be known only at runtime; the composed service must also support the REST uniform interface; data types are known at runtime; content negotiation must be allowed; and clients must be able to inspect the composed service [16], [17].

Currently there are no proposals for automatic and dynamic composition of REST services. REST services composition research is mainly focused on static service composition [16], [17]. The most comprehensive work in the area is the JOpera project [18] that aims to provide a similar implementation of BPEL for REST. JOpera allows the static composition of Web services by means of two graphs. The first, Flow Control Dependency Graph (CFDG) describes the sequence of invocation of services, and the second, the Data Transfer Flow Graph (DFTG) defines the relationship between the data inputs and outputs when invoking the REST components. Unlike BPEL, service composition in JOpera allows that the service URI to be invoked is known at runtime (dynamic late binding), it also suports the uniform interface, and content negotiation at

runtime. However, JOpera does not consider the HATEOAS constraint, and the CFDG is performed by a centralized process engine, in a stateful fashion which negatively impacts on service scalability.

Similarly, Bite [19] proposed a BPEL-inspired composition language describing data and control flow. Bite partially supports a uniform interface based on HTTP, dynamic data types, and state inspection (only `GET` and `POST`). Regarding dynamic late binding, Bite can generate URIs to created resources, but cannot inspect the service's responses and find out the links provided by the service (HATEOAS).

Decker [20] presents a formal model for implementing REST processes based on Petri nets, they uses PNML (Petri Net Markup Language) as a language for specifying the states, transitions and an execution engine. Decker considers only partial support for dynamic late binding since it can generate URIs for resources created but cannot inspect the responses and retrieve the embedded links. In this model, control flow is driven by the decisions of a human user. Guard conditions, such as autentication are not supported and like others [21] it assumes XML as the content type for representations leaving out REST content negotiation constraint.

Zhao [22] presents an automatic service composition approach for REST typifying and semantically describing the services in three categories according to the operations that the service can perform (`GET`, `PUT`, `POST`, `DELETE`). Service composition is automated by a first-order logic situation calculus representing changes and actions. This approach does not consider several of the REST principles such as HATEOAS, content negotiation, opaque URIs, nor dynamic late binding.

2.5 QoS

The quality of service (QoS) is a combination of several qualities or properties of a service [23]. Dynamic composition of Web services requires the consumer to choose the best component services that satisfy the functional and non-functional requirements of the composition. Non-functional requirements involve attributes of quality of service as response time, availability, security and performance [23].

The definition and measurement of these attributes of service quality vary by approach. For example, the response time can be measured as the average of the results obtained of the last 15 minutes, or an average vector of 15-minute intervals during the day.

The quality attributes of a service depend on the load current at which the service is submitted, however the technical description of these quality attributes of a service are focused on the description of a discrete value associated with each property.

QoS in REST

For the case of REST, research initiatives on service composition are fairly recent, and interest on QoS properties have focused mainly on security. For instance, in [24] a RESTful service API is defined to support service-level agreements based on the WS-Agreement standard. Agreements (and templates) are REST resources encoded in the application/xml media-type, whose life cycle and state is handled by means of HTTP verbs. Graf et al. present a server-side authorization framework based on rules that limit access to the served resources according to HTTP operations [25]. In [26] a server-side obligation framework allows designers to extended existent policies with rules regulating users' information access. Rules may trigger additional transactions (e.g., sending a confirmation e-mail, register the information access attempt in a log) or even modify the response content or the communication protocol (e.g., require HTTPS). Allam [27] aims to homologate WSDL-based and RESTful services by considering them black boxes, where interaction occurs as simple message passing between clients and servers. Security policies can be placed when receiving or sending a message as well as locally (e.g., at the server or client side). This vision does not consider complex interaction involving third parties (e.g., OAuth), or service's interface heterogeneity [28], where industry standards are implemented in various ways. We assume a workflow perspective where data is transformed locally in successive steps until the message constraints required by the service provider are achieved. In addition, clients, servers, and third party services may engage in an interaction that implements sub-workflows.

3 REST LIMITATIONS

The REST architectural style offers several advantages when compared to traditional Web services

in terms of non-functional attributes, namely, low latency (small response time), massive scalability (due mainly to statelessness, cacheability, and Web intermediaries), modifiability, and simplicity, among others. However, REST considers humans as its principal consumer and they are expected to drive service discovery and state transition by understanding the representation content. The lack of a REST machine-readable description forces service providers to describe their APIs in natural language, which makes difficult to properly design machine-clients that can perform discovery and service composition in an automated way.

Furthermore, REST has been used in the industry as an infrastructure layer for supporting massive service provisioning in the form of Web APIs which have given rise to the evolution of application ecosystems that constitutes simple service composition basically consuming services in a sequential control-flow pattern. A massively used example of complex control-flow is the OAuth protocol that constitutes an orchestration where various parties cooperate through redirections in order to implement a workflow. The control-flow patterns involved in the OAuth are sequence and conditional execution. Researchers also recognize the lack of a complex service behavior model in REST as one of the difficult issues to be addressed in order for REST to support rich SOA (Issarny, 2009). Research proposals for REST service composition focus either on operation-centric [29], centralized [7], stateful and static service composition, violating REST architectural constraints.

Traditional Web services dynamic and automatic composition focuses on diverse techniques (e.g. planning, graph models, semantic models, QoS constraints, etc. in order to make possible the automatic or dynamic generation of a composition plan (i.e. a workflow) and the selection of the services components [30]. QoS has been a focus of extensive research and is being recently addressed in REST but not in its capacity for determining service composition on runtime. Furthermore, the very nature of the non-functional attributes makes it hard to reduce them to uniform representations such as numbers or Boolean values. Security for instance requires not only a complex combination of algorithms but also protocols (in the case of OAuth a choreography) in order to determine whether a service can be composed or not. For the case of response time, availability and throughput, research in Web sites provisioning, also known as capacity planning, have demonstrated the variables such as the user demand and cache policies are relevant to determine the quality of a Web site (equivalent to a single REST service). Hence such quality attributes cannot be reduced to simple numeric variables as is the case of traditional QoS-aware compositions in SOA.

4 General Goals

The overall objective of this thesis is to facilitate the dynamic composition of REST services. In particular, we propose:

1) A decentralized RESTful, stateless composition model that places an emphasis on service behavior (control-flow),
2) A QoS model focused on security in order to determine the feasibility of composition at runtime, and
3) A QoS model focused on scalability, throughput and response time in order to select the service component automatically at runtime.

5 Hypothesis

The main hypothesis of this thesis is related to the relevance of decentralized service composition, in that *"a stateless and decentralized composition technique follows the REST architectural style constraints and generates a composite service with the same REST properties whereas a centralized composition does not"*. Second, *"QoS-based dynamic and automatic RESTful service composition must take into account the characteristics of the non-functional attributes in order to preserve REST architectural constraints in the composite service"*.

6 Thesis Work and Main Contributions

The first part of this thesis presents a technique for decentralized, hypermedia-centric and stateless REST services composition. The proposed technique models services behavior through a set of well-known, simple and complex control-flow patterns and focuses on the following REST constraints, namely, client-server interaction, layered client-server, client-stateless-server, and the uniform interface. The uniform interface (messages between server and client are self-descriptive) was extended

through `300 HTTP` redirection codes in order to include control-flow information in the message; hypermedia (Web linking) is used as the application state machine. We present the advantages of this approach (centralized Vs. decentralized) in terms of response time, availability and throughput, which are non-functional goals in REST. By definition a stateful approach is not RESTful so that such comparison is left out.

The second part of this thesis describes automatic and dynamic REST services composition based on non-functional attributes. In this part two QoS domains are analyzed, the first corresponds to security and the proposal is a hybrid between static and dynamic service composition in the sense that a service description (created at design-time) is used to determine the feasibility of service composition (at runtime) and actually enforce the composition (at runtime). The environment is decentralized, stateless and promotes the use of hypermedia as a state machine.

The third part of the thesis focuses on a second QoS domain considering scalability. That is, the non-functional attributes: response-time, throughput and availability were used to support automatic and dynamic service composition. In this case, a queuing theory-based model was used to identify response-time, throughput and availability. These models were later used in a technique called SAW-Q to identify the compositions with the highest quality. SAW-Q was compared with a well-known technique, SAW (from which SAW-Q was derived) with good results.

Accordingly, the main contributions of this thesis are:

1) Part 1
   a) A decentralized, stateless, hypermedia-centric technique for designing composed service behavior in REST.
   b) A set of control-flow patterns that implement decentralized, stateless, hypermedia-centric REST service composition.

   These contributions were published in the following journal:
   J. Bellido, R. Alarcon and C. Pautasso. *Control-Flow Patterns for Decentralized RESTful Service Composition*. ACM Transactions on the Web 8:1 (5:1âĂŞ5:30). ACM 2013.
2) Part 2
   a) A centralized, hybrid (design-time and runtime) and manual REST service composition based on the security QoS attribute.
   b) A security domain model.
   c) An extension to ReLL, a hypermedia REST service description, that considers the security domain model in order to determine the feasibility of consuming a protected service and actually execute the OAuth choreography:

   These contributions were published in the following journal:
   C. Sepulveda, R. Alarcon, and J. Bellido. *QoS aware descriptions for RESTful service composition: security domain*. World Wide Web 1-28, 2014
3) Part 3
   a) A decentralized, dynamic REST service composition technique based on the response-time, throughput and availability QoS attributes.
   b) A queuing theory-based REST model.
   c) A response-time, throughput and availability models based on the proposed queuing model.
   d) SAW-Q, a refinement of SAW, a technique for scoring service compositions in terms of various variables such as response-time, throughput and availability that is sensible to user demand and service capacity.

   These contributions were submitted to the following journal:
   J. Bellido, R. Alarcon, and C. Pautasso. *SAW-Q: An approach for dynamic composition of REST services*. IEEE Transactions on Services Computing, 2014.

Other articles were also produced during this investigation:

- R. Alarcon, E. Wilde and J.Bellido. *Hypermedia-driven RESTful service composition*. Service-Oriented Computing. Springer Berlin Heidelberg, 2011. 111- 120.
- J.Bellido, R. Alarcon and C. Sepulveda. *Web Linking-based protocols for guiding RESTful M2M interaction*. Current Trends in Web En-

gineering (pp. 74-85). Springer Berlin Heidelberg, 2012 15

The remainder of this document is organized as follows: Section 7 presents the journal that summarizes the results of the first part of the thesis. Section 3 explores the attributes of service quality in the RESTful services composition, focused on security. Section 8 presents the dynamic composition approach based on SAW-Q and scalability (response-time, throughput, availability). Section 8 presents the main conclusion of this thesis and future research.

7 CONTROL FLOW PATTERNS FOR DECENTRALIZED RESTFUL SERVICE COMPOSITION

Describes a technique for decentralized REST services composition that takes into account the constraints of REST architectural style in the composition process. The proposed technique involves the creation of control-flow patterns that allow seamless interaction between the client and the composite service and uses hypermedia as a state machine. The main idea is to implement control-flow patterns through callbacks and redirections. Finally, we discuss the impact of our design decisions according to the architectural principles of REST.

The redirection code (`303`) was designed to inform the user agent it must fetch another resource, and it is widely used for services to interact with other services and accomplish business goals. For example, OAuth and OpenID are popular authorization and identity protocols implemented using redirection codes; payment entities which offer online transactions are also implemented using redirection codes in order to allow e-commerce applications to sell products online in a security context under their control.

Due to constraints on the `303` redirection code, it cannot support complex interaction successfully. For instance, parameters should be serialized in a text format and concatenated to the URI (`application/x-www-form-urlencoded`), and information that cannot be serialized as plain text cannot be passed between applications in the URI parameters (e.g., images, pdf documents, etc). The resulting URI must not exceed the limit established by the server, otherwise the server should return a `414 Request-URI Too Long` status code message. In order to send large quantities of data, the media type `multipart/form-data` and the `POST` method shall be used for submitting forms that contain files, non-ASCII data, and binary data. In addition, only the `GET HTTP` method can be used to automatically fetch the redirected URI, but as seen in our example, applications may be required to interact with each other using additional methods without requiring end-user confirmation (e.g., `POST` and `PUT` messages 3 and 10 in Figure**??**).

More importantly, control flow may be more complex than sequential invocation of REST resources. Business processes also require parallel or alternative invocation as well as determining the conditions for choosing the right response; more complex control flows consider the invocation of two services in non established order but only one at a time (unordered sequence), or service invocation for a determined number of times iterator.

Finally, notice that the composed REST service (`PO` and `/stateN` resources) encapsulates the knowledge about which services to invoke (URI), which parameters or state information should be sent and be expected to be received, which methods shall be used (e.g., `GET` (7) or `POST` (12) in Figure **??**), as well as the order of the invocation; that is, they must know the service interface of the resource, which in our case is accomplished through ReLL.

7.1 Control flow patterns

In the context of stateless, decentralized compositions of services described with ReLL and with the assumption that clients can process the Callback link header, in this section we model control-flow patterns for RESTful service composition and the `HTTP` protocol. The set of patterns includes sequence, unordered sequence invocation, alternative, iteration, parallel, discriminator, and selection [31], [32].

Sequence, Unordered Sequence

The sequence pattern is a basic control-flow structure used to define consecutive invocations of services which are executed one after the other without any guard condition associated. As seen in figure **??**, this pattern could be implemented using the `303` redirection code; however, only automatic redirection of `GET` messages are allowed by the standard,

making it difficult to update the composed resource state (i.e., `PUT` message of lines 5, 9). In addition, there is no clear indication on how to handle the payload of the message. We extend the status codes with a set of codes identified with a two digit prefix: `31x`. The sequence pattern is implemented with a new code: `311 (Sequence)` indicating the invocation of a service without any guard condition (see Figure 1).

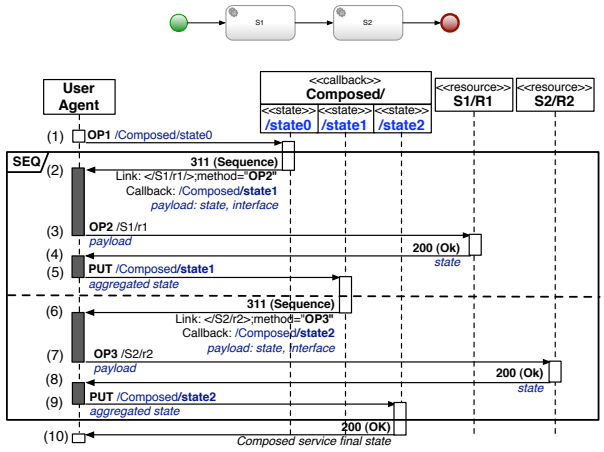


Fig. 1. A sequence control flow pattern implemented for REST and HTTP

The server responds with a `311` message including the component resource address (2, 6) in a `Link` header as well as the HTTP method in a link target attribute, and a `Callback` address in an additional header indicating the state of the composition. Additional information such as state (e.g., a digested value) and, depending on the service interface, data formats or URI schemes to create the request, can be included in the payload. Actual data values for such templates shall be provided by the user agent either by requesting them to the user through an appropriate interface or retrieving them from the local storage. Such process is out of the scope of this proposal. The server may close the connection with the client after issuing a `311` message unless metadata indicating otherwise is included. When a user agent receives this code, it must store locally the callback address and automatically request the component resource using the indicated method (3, 7). Similarly to Figure **??**, if additional communication shall occur between the component resource and the user agent it must be modeled as out-of-band communication and is omitted for readability. When the response is available, the component replies with a `200` status code. The composed service shall not issue another request until the response has been passed by the user agent through a `PUT` message (5), then the composed service can proceed with the next component (6 to 9).

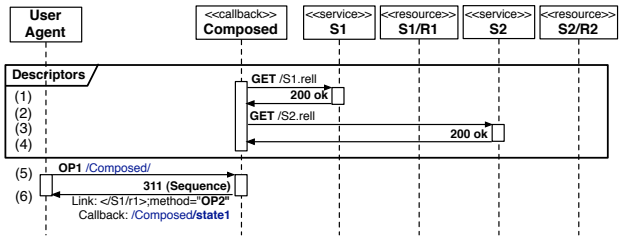


Fig. 2. ReLL descriptors are fetched considering the root resource of a service

When all the components have been fetched (i.e., the final state of the sequence has been reached), the response is provided with a `200` status code and the composed service representation (10). Notice that the actual HTTP methods to be used when invoking component services must be determined by the composed resource. In order to know how to handle the resources, the composed service prefetches the component services descriptors which detail the interface of a set of resources at domain-level; the descriptors are themselves REST resources (Figure 2). This phase is omitted in the figures detailing the remaining patterns for readability, although it is assumed it takes place before invoking a component resource.

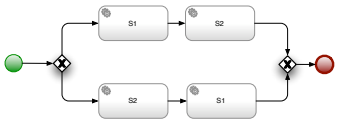


Fig. 3. Unordered Sequence

For the case of the unordered sequence pattern, it specifies the execution of two or more services in any order but not concurrently (Figure 3). That is, given two services `S1` and `S2`, the services execution can result as `S1` followed by `S2` or `S2` followed by `S1`. Since the list of services to be invoked is known by the composed resource and the order is irrelevant, the composed resource (server) has the information to decide which service to invoke as part of its own logic. For the user agent, all that matters is the address of a particular component resource to

be invoked as well as the method; that is, this case is not different from a sequential invocation.

Alternative, Exclusive choice

The alternative pattern is a basic control-flow structure that allows the execution of only one of two possible services. The service to be executed could depend on the outcome of preceding service execution, values of the parameters, or a user decision. The `312 (Alternative)` status code is proposed for this pattern. When a composed service requires executing one of two services, it responds to the client request with a `312` coded message, indicating the list of services to choose as `Link` headers, including the HTTP `method` as an attribute, and a `Callback` header indicating the connector state to resume interaction. The message payload is a conditional expression to be evaluated by the user agent, as well as information required to build proper request messages (i.e., data formats or URI schemes).

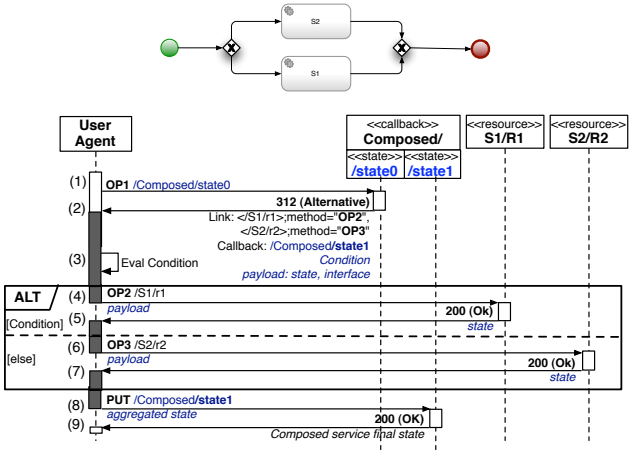


Fig. 4. An alternate control flow pattern implemented for REST and HTTP

The composed resource closes the connection after issuing the response unless otherwise indicated by additional headers. Link services may differ on the resources (URIs), or the methods to be used (Figure 4, message 2). Since in REST user agents keep application-state information [33], they shall have enough information to perform the evaluation. A good practice is to express the condition in languages well-known to the Web, such as XPath, although its format escapes the scope of this thesis. Once the user agent has evaluated the condition it determines which link to follow (4 or 6). Again, additional communication may occur between a user agent and an origin server. Eventually when the component has a final response, it issues a `200` coded response including its state in the payload (5 or 7). This causes the user-agent to send an update message (`PUT`) to the composed resource carrying on the received payload (8). Once the interaction finishes, the composed resource replies with a `200` message including the representation of its final state (9).

Iteration, Structured Loop - while variant

This pattern is an advanced control-flow structure that allows the execution of a service repeatedly, the number of times depending on a fixed number, a time frame, etc. which can be modeled by a conditional expression. We propose the `313 (Iteration)` status code for representing iterations.

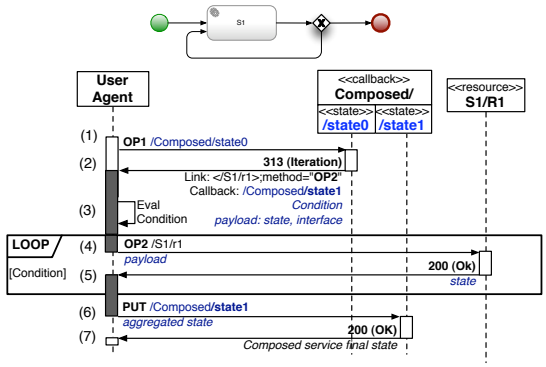


Fig. 5. An iterator control flow pattern implemented for REST and HTTP

This interaction begins when the composed resource issues a `313` message (Figure 5, message 2) including a `Link` header with the address of the component resource, a `Callback` header indicating the callback connector state address, a conditional expression, and additional information to create the message request as part of the payload. After evaluating the conditional expression (3) and obtaining positive results, the message is redirected to the component resource using the indicated operation and payload (4). Communication between client and server may include several messages interchanged. When a response is available, the component resource will issue a `200` message (5). The condition will then be evaluated again. If it still holds, the component is invoked again (4);

or a `PUT` message is sent to the callback address carrying along the response content served by the component service aggregated with previous state information (6). Finally, at the end of the interaction, the component replies with a `200` message and the representation of the composed resource final state (7).

Parallel Split - Synchronization, Structured Discriminator, Structured Partial Join, Local Synchronization Merge (Selection)

The Parallel Split is a simple pattern that allows a single thread of execution to be split into two or more branches invoking services concurrently. The parallel split pattern can be paired with either one of four advanced control-flow structures. Under the paradigm of a composed service - component services, it is the former which determines whether it waits for all the responses (Synchronization, Figure 6.a), just one of them (Structured Discriminator, Figure 6.b), or a fixed number (Structured Partial Join, Figure 6.c). Finally, for the case of Local Synchronization Merge (also called Selection, Figure 6.d), the composed service shall wait for a number of responses that cannot be determined with local information.

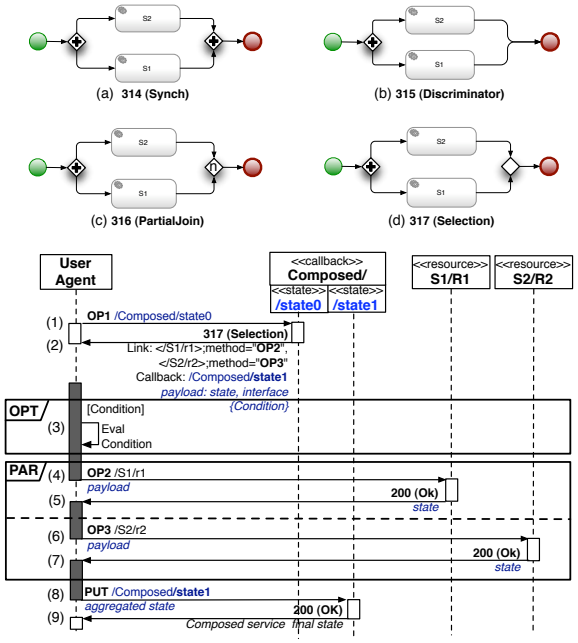


Fig. 6. A parallel control flow pattern implemented for REST and HTTP

In order to avoid violating the REST stateless principle, servers do not store information about how many answers are expected per client, but make explicit server's expectancies through the pattern status codes, `314 Synch` (Synchronization), `315 Discriminate` (Structured discriminator), `316 PartialJoin` (Structured Partial Join) and `317 Selection` (Local Synchronization Merge). The four patterns follow the same conversational structure; however, the client's decision to inform the server about the availability of a final response is affected by the corresponding pattern.

Figure 6 shows the details for the pattern. Interaction starts when the composed resource issues either a `314`, `315`, `316` or `317` message (Figure 6, message 2). The message includes a list of `Link` headers annotated with a `method` attribute, a `Callback` header indicating the callback connector state address, and a payload with instructions to format input data for the operations according service interface. It may also include state information such as the number of expected service components to be addressed by the client for the case of the `316 Partial Join` pattern.

For the case of a `317 Selection` message, a conditional expression must be included. The condition must be evaluated considering application-state information stored locally at the client side (3), and the result shall be the number of request messages the client must issue to service components.

Once the user agent determines how many responses to provide to the composed resource (all, one, n out of m, or n), it invokes all the service components indicated in the list with the appropriate methods concurrently (4, 6). In practice the number of links to be fetched is limited by the number of concurrent connections the client is able to maintain with the servers involved. Again, there may occur several messages interchanged between clients and origin servers as an out-of-band conversation, but once the final response is available, it is aggregated until the number of responses expected to be sent to the composed service is reached. The aggregated state is sent as a `200` coded request (8). The composed resource processes the aggregated state (e.g., it could merge the results) and issues a `200` coded response with the final state.

8 QoS AWARE DESCRIPTIONS FOR RESTFUL SERVICE COMPOSITION

We explore QoS aware RESTful services composition, which is characterized by a decentralized, stateless and hypermedia-driven environment. We focus particularly on the security domain since current security practices on the Web illustrate the differences between both the centralized, function-based approach and the decentralized, hypermedia and resource-based approach. We rely on ReLL (a REST service description) that can be processed by machine-clients in order to interact with RESTful services. Our approach identifies key security domain elements as an ontology. Elements serve to model hypermedia-based, decentralized security descriptions supporting simple and complex interaction such as protocols and callbacks. In this paper, we propose an extension to ReLL that considers security constraints (ReLL-S) and allows a machine-client to interact with secured resources, where security conditions may change dynamically. A case study illustrates our approach.

8.1 ReLL-S security description

Compared to traditional services, security is addressed in a different way on the Web. In Maleshkova et al. [34] a review of the self-declared REST Web APIs corresponding to the ProgrammableWeb site [2] was analyzed regarding their support of security mechanisms. The Maleshkova et al. study analyzes a Web site that is a well-known repository for services (including WSDL, REST, XML-based, Web APIs, etc.). They picked 222 services from the RESTful service category (18%) covering the 51 service categories (e.g., search, geolocalization, etc.) available. The security mechanisms found range from very simple practices (the majority), to the W3C standards on security on the Web. For instance, 38% uses API Keys whilst the OAuth protocol is used by 6% of the reported service. Notice that mainstream so-called REST services (e.g., Facebook, Twitter, Google, etc.) support and require OAuth authentication. Therefore, we believe the study is representative from a practical (and informal) and a theoretical (standards) point of view. In this section we present ReLL-S descriptions supporting each of the security mechanisms identified in the Maleshkova et al. survey.

2. http://www.programmableweb.com

8.2 OAuth

OAuth [35] is an Open Authorization protocol that allows a third-party application - a client application - to access resources provided by a service - resource server - and owned by a user. The user has to authorize the third-party application to access the resources stored by the resource server, without exposing his or her authentication credentials, to the third-party application. The authorization grant is represented as a token.

OAuth defines four grant types: authorization code, implicit, resource owner password credentials, and client credentials; and provides an extension mechanism for defining additional grant types. The protocol flow is flexible and depends on the type of authorization that is going to be granted. So in the flow shown in Figure 7, up to four parties could be involved: the resource owner, the resource server, the authorization server, and the client. The result of the protocol is an access token that represents the authorization granted by the resource owner and that is sent later by the client to the resource server to access the restricted resources.

What is interesting about this protocol is that it involves more than two entities that can communicate using a protocol that has been designed for client-server communication, as it is HTTP, in a stateless one-to-one conversation. However, this characteristic presents particular issues to deal with, which makes it really interesting to describe in more detail.

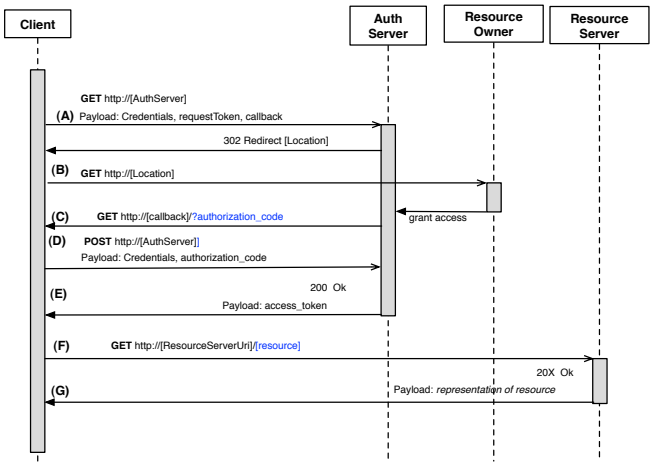


Fig. 7. Abstract OAuth interaction between four parties

Once the client has required the access grant, the

authorization server starts a conversation with the resource owner that, from the client's perspective, occurs completely out of its control. This conversation's goal is asking the user to give authorization to access the resources. It could be implemented many ways; the authorization server could send an email to the user, an SMS, or any other kind of conversation. In most implementations the conversation occurs synchronously by redirecting the user-agent from the client domain to the authorization server domain. So that, to restore the interaction between user and client, the authorization server must also redirect the user-agent to the client. If the user grants access rights to the client, such redirection message will contain an authorization grant. This conversation implements an asynchronous conversation through a callback connector provided by the client, which becomes the target of the redirection.

Once the client has the authorization grant, it can use it to retrieve an access token from the server. The client sends its credentials and the authorization grant, representing the permissions granted by the user, to the server. The server verifies the permission and generates a final token (access token, or oath token) to be used in future calls, so that the resource server allows access to restricted resources. It could have an expiration time and some authorization servers provide the feature to refresh or renew the token.

Listing 1. An authorization constraint specification (OAuth)

```
<scope resource="restrictedResources">
  <constraint id="oauthAuth"
     type="Authorization">
    <accessToken>
      <query_param name="auth_token"
         select="$auth_token||wl:OAuth"/>
    </accessToken>
  </constraint>
</scope>

<protocol name="wl:OAuth">
  <invoke
     url="http://www.authserver.com/oauth"
     pre="not($auth_code)">
    <query_param select="$state"
       name="extra"/>
    <query_param select="$callback"
       name="callback"/>
  </invoke>
  <invoke
     url="http://api.service.com/getToken"
     pre="$auth_code">
    <query_param select="$auth_code"
       name="auth_code"/>
    <store selector="token"
       persist="auth_token"/>
  </invoke>
</protocol>
```

In this case the client makes two calls to get authorization to the user's restricted resources. The first one could include the callback address and also a parameter, named `state`, that must be sent back by the authorization server when issuing the callback request to the client. The client can use the parameter to keep the flow state. The callback call will include also the authorization code used by the client to get the final authorization token during a second call. The order of both calls is defined by their preconditions (i.e. they play the roles of guard conditions). That is, when the client runs the protocol, it will invoke the first call initially because it doesn't have the `auth_code` stored locally. Once the request is issued, the client blocks itself waiting for an answer. When the callback arrives, the client is restarted and since its current condition has changed (i.e., it has the code) then it will make the second call (the condition is met) to finally resolve the authorization constraint.

9 SAW-Q: An approach for dynamic composition of REST services

SAW-Q: An approach for dynamic composition of REST services: We propose SAW-Q an extension of SAW, as a technique of dynamic composition according to the principles of the REST style and considering quality attributes modeled as a demand function instead of the traditional constant values. Our model is much more accurate when compared to real implementation, positively improving dynamic composition.

9.1 Modeling REST QoS service using Queue Models

There are multiple ways to measure the performance of a service. The most common metrics are response time, availability and throughput. A service response time is the time interval from the request arrival to the server until the response is received by the

client and is determined by two factors: the network latency and bandwidth and the service capacity. The way a REST service processes requests and replies can be modeled using queuing theory. The client requests follow the Poisson distribution [36], [37] and are randomly distributed over a period of time. The generalized queuing model considers the long-term behavior of the queue or steady state, reached after the system has been running for a sufficiently long period of time.

Performance measures of a queue are based on the probability that a certain number of requests may exist in a system at a given time (p_n). For instance, p_0 would represent the probability of a minimal waiting time due to an empty queue (0). More in detail, useful performance measures are[37]: the expected amount of requests in the system (L_s), which includes the expected number of requests from customers in the queue (L_q), the waiting time of a request to be processed by the service (W_s) and the waiting time of the request in the queue (W_q).

Some specializations of the generalized queue system that calculates performance measures have been proposed and proved. The specialized queue model corresponding to a REST service is $M/M/C : GD/N/INF$, where $N \geq C$ as described before. The corresponding performance measure formulas originally defined in [37] are described below.

Given the estimated rate of requests (λ) that arrive at the system (S), the rate of denied request depends on the rate of arrival at a given time (Formula 1). The higher the arrival, the higher the probability of denied requests.

$$\lambda_{lost} = \lambda \cdot p_N \tag{1}$$

The effective rate of arrival is the difference between the arrived requests minus the denied requests (Formula 2)

$$\lambda_{eff} = \lambda - \lambda_{lost} = (1 - p_N) \cdot \lambda \tag{2}$$

The rate between arrival (λ) and processed requests rate of the system (μ) is defined by $\rho = \lambda/\mu$. It is possible to calculate the expected number of requests waiting in the queue $L_q(s)$ using Formula 3 [37] , and the expected number of requests in the $L_s(s)$ system using Formula 4.

$$L_q(s) = \frac{\rho^{c+1}}{(c-1)!(c-\rho)^2}\left\{1 - \left(\frac{\rho}{c}\right)^{N-c+1} - (N-c+1)\left(1\right.\right. \tag{3}$$

$$L_s(s) = L_q(S) + \rho \tag{4}$$

The processing time for a REST service is defined as a rate between the number of processed request and the effective arrival rate of requests (Formula 5).

$$W_s(s) = \frac{L_s(s)}{\lambda_{eff}} \tag{5}$$

The waiting time in the waiting queue for a request in a REST system is defined as a rate between the number of requests waiting in the queue and the effective arrival rate of requests (Formula 6).

$$W_q(s) = \frac{L_q(s)}{\lambda_{eff}} \tag{6}$$

The average service response time $Q_{ResponseTime}(s)$ is calculated as the sum of the processing time of the services (W_s) and the waiting time in the queue of requests (W_q), as seen in Formula 7. We are not considering network latency since this is out of the scope of responsibility and control of the service.

$$Q_{ResponseTime}(s) = W_s(s) + W_q(s) = \frac{L_q(s)}{\lambda_{eff}} + \frac{L_s(s)}{\lambda_{eff}} \tag{7}$$

Replacing the factors in the formula in order to reflect the clients demand dependency, we obtain the following Formula 8.

$$Q_{ResponseTime}(s) = \frac{L_q(s)}{(1-p_N)\lambda} + \frac{L_s(s)}{(1-p_N)\lambda} \tag{8}$$

Service processing capacity is defined as the number of requests that a system (the whole number of service replicas) can process at a given time. The service availability $Q_{Availability}(s)$ is usually measured as the percentage of time that the service is ready to be consumed at a period of time [37]. However, the availability of a service depends on the number of customers attempting to access a service. If the number of requests (n) exceeds the capacity of the service ($C \leq n < N$), the availability is downgraded because many request try to access and compete for the service resources, hence the

probability of being served is reduced. Otherwise, if the number of requests (n) falls below the service capacity ($0 \leq n < C$), the availability of the service grows. Service availability is calculated as the probability that n requests exist in the system (p_n). Service availability is calculated using Formula 9.

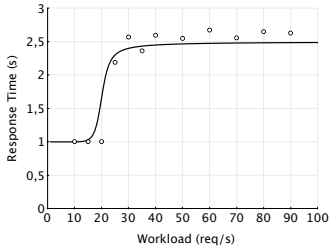
$$Q_{Availability}(S) = \begin{cases} \frac{\rho^n}{n!} p_0, & 0 \leq n < c \\ \\ \frac{\rho^n}{c! c^{n-c}} p_0, & c \leq n < N \end{cases} \quad (9)$$

The service throughput $Q_{Throughput}(s)$ is the number of requests that the service can process in a unit of time (i.e. seconds). If $Q_{ResponseTime}$ is the function to calculate the time that takes a request to be processed by the service, then the quantity of requests that can be processed by the service in one second is calculated as the inverse function of $Q_{ResponseTime}$, as seen in Formula 10 .

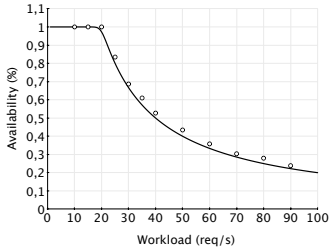
$$Q_{Throughput}(S) = \frac{(1 - p_N)\lambda}{L_q(S) + L_s(S)} \quad (10)$$

9.2 Applying the Queue Model in a real scenario

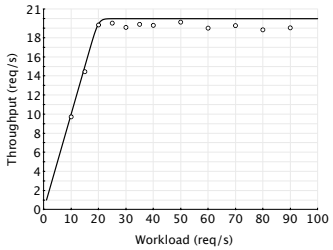
We implemented a REST service as a Python script running on a Apache Web server. The experiment was performed locally, in order to discard the effect of network latency, on an Intel PC with 4GB RAM, and 4 cores. We also modeled response time, availability and throughput using the equations proposed in section 3.4. The implemented service is characterized as $W_s = (1, 20, 50)$ where each instance is able to process one request per second, the service is replicated 20 times and there may be up to 50 requests in the system at a given time. The response time was calculated using the proposed formulas and we obtained the experimental results from the stress tests using Apache JMeter, the tests were run 10 times automatically and averaged, in order to eliminate external factors. For each scenario, results between the models and the experiments are very similar, for the case of response time the results show that when the number of requests exceeds the processing capacity of the service, the response time increases until the requests are denied (Figure 8(a)); service availability decreases exponentially (Figure 8(b)), and throughput remains constant once the service reaches the maximum capacity (Figure 8(c)).



(a) Response Time



(b) Availability



(c) Throughput

Fig. 8. Comparison between estimated (continuous line), using the proposed formulas, and actual (shown as dots) behavior of a service in different scenarios. (a) The average response time of a service increases when the workload exceeds the capacity of the service (20 req/s); (b) service availability is reduced when the service is unable to process more requests (20 req/s); (c) and service throughput grows according to the workload until reaches its maximum capacity (20 req/s).

9.3 Dynamic Composition based in QoS using Queue Model

In order to compose a service dynamically and based on the quality attributes, we need to select the service components as late as possible (dynamic late binding) and at runtime. In this paper we propose a strategy of hybrid dynamic composition that combines the techniques of workflow driven composition and declarative composition. Composition techniques guided by a workflow define an abstract process model for the composed service and

later each activity of the model is bound to a particular service. Declarative composition techniques, on the other hand, use customer-defined rules and constraints to determine if the resulting composite service meets customer expectations. In this case, service selection will depend on the fulfillment of each user-defined condition. Quality attributes of the resulting concrete service composition can be evaluated using a utility function using global and local optimization techniques [38].

On the other hand, the workflow can be seen as an arrangement of control-flow operators (e.g. sequence, conditional, parallel, etc.) that impact the utility function defined in a declarative technique. For example, the response time of a composed service that includes the sequential invocation of service components, results in the sum of the response time of the service components. If the service invocation happens in parallel, the response time of the composed service is the maximum value of the response time of the invoked services. When the invocation occurs within a loop, the composed service response time is the product of service component response time and the times that it is invoked. For the case of the conditional control-flow pattern, the response time of the composed service, in the worst case, is the maximum response time of the components to be chosen by the alternative condition. In [38], the influence of control-flow operators in the quality attributes of composed services are defined as aggregation functions.

A typical example of service composition is the Travel Planner. In this composite service, the search of places for entertainment (Attraction search) is performed in parallel with the reservation service and an airline reservation service of a hotel. Then, the distance between the place of entertainment and the hotel is calculated and depending on the outcome, a car rental or a bike rental service may be chosen. The abstract process of this composite service can be seen in Figure 9.

According to the algebra of service composition [39], the Travel Planner service is defined by $W_{tp} = \{(AttractionSearch \| (TicketBooking \odot Hotelbooking)) \odot DriveTimeCalculation \odot (BikeRental \diamond CarRental)\}$. In the notation, control-flow dependencies are denoted by symbols, for instance $\|$ defines parallel invocation, $\odot$ defines sequential invocation, and $\diamond$ defines a conditional invocation. Each task of an abstract process can be implemented using one of many existing services, which results in a composite service that should meet the user expectations. The number of possible ways for composing this Travel Planner service depends on the number of service candidates for each task. That is, if each task in the abstract process has two service candidates, then there are 2^6 different ways to implement the composite service. The objective of dynamic composition is to find the combination that gives the user the greatest benefits. In the remainder of this section we introduce SAW, which proposes a utility function that identifies the best combination according to multiple criteria.

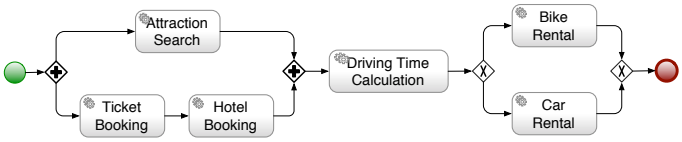


Fig. 9. Abstract BPMN model of the composite service travel planner

Simple Additive Weighting (SAW)

One way to assign a score to each possible service combination is by using the Simple Additive Weighting (SAW) utility function defined by Formula 13 [40]. The SAW technique consists of a process of assigning scores to each combination through two phases: scaling and aggregation or weighting. At the stage of scaling the values of the quality attributes of the candidate services are normalized between 0 and 1 by a comparison between the maximum and minimum values. Then, during the aggregation phase the following formulas are used to calculate the score of a composite service.

To define a particular composed service, we consider an abstract model $CS_{abstract} = S_1, S_2, ..., S_n$, restricted by customer defined constraints $QoS_{constraints} = C1, C2, ..., Cm$. The process that assigns a score for each combination of services $CS_x = s_1, s_2, ..., s_n$ of the abstract model works as follows: the quality attributes of each candidate service s is represented by the feature vector $q(s) = q_1, q_2, ..., q_k$, therefore $q_k(s)$ represents the k-quality attribute value for service s. Thus, $Q_{min}(j,k)$ and $Q_{max}(j,k)$ defined in Formula 11 represent the minimum and maximum value of the k quality attribute for the service candidates to implement a service S_j task.

$$Q_{min}(j,k) = \min_{\forall s \in S_j} q_k(s)$$
$$Q_{max}(j,k) = \max_{\forall s \in S_j} q_k(s) \quad (11)$$

Then $Q'_{min}(k)$ and $Q'_{max}(k)$ defined in Formula 12 calculate the minimum and maximum value of the k quality attribute for the abstract model using the aggregation functions F, defined in Table **??** for each operator of the model.

$$Q'_{min}(k) = F^{n}_{k\,j=1}(Q_{min}(j,k))$$
$$Q'_{max}(k) = F^{n}_{k\,j=1}(Q_{max}(j,k)) \quad (12)$$

Finally, the utility function assigns a score to the composite service CS_x, calculated by Formula 13 where the scores obtained for each quality attribute k, are weighted by the user preferences $W = (w_1, w_2, \ldots, w_k)$

$$U'(CS_x) = \sum_{k=1}^{r} \left(\frac{Q'_{max}(k) - q'_k(CS_x)}{Q'_{max}(k) - Q'_{min}(k)} \cdot w_k\right) \quad (13)$$

Simple Additive Weighting using Queue Model (SAW-Q)

Since dynamic composition aims to find the best service for each task at runtime, the time required to meet this objective has an important role. It is for this reason that techniques such as integer programming, dynamic programming, heuristic and distributed processing are mainly used in order to reduce the time service selection. However, most techniques ignore service demand while evaluating quality scores, which causes that the resulting composition may not be the most useful in practice. Hence, we modify the SAW technique to take into account the expected customer demand.

Unlike the previous model, in this model each service candidate s is defined by the vector of quality attributes $q(s,\lambda) = q_{rt}(s,\lambda), q_{av}(s,\lambda), q_{th}(s,\lambda)$. That is, by modeling REST services as a queuing system, we can calculate the quality attributes of a service according to the expected customer demand for the composite service (λ), using the formulas 8, 9 and 10 (Section 9.1). Hence, the SAW formulas presented before are modified introducing the user demand:

$$Q_{min}(j,k,\lambda) = \min_{\forall s \in S_j} q_k(s,\lambda)$$
$$Q_{max}(j,k,\lambda) = \max_{\forall s \in S_j} q_k(s,\lambda) \quad (14)$$

$$Q'_{min}(k,\lambda) = F^{n}_{k\,j=1}(Q_{min}(j,k,\lambda))$$
$$Q'_{max}(k,\lambda) = F^{n}_{k\,j=1}(Q_{max}(j,k,\lambda)) \quad (15)$$

The proposed utility function including clients' demand, SAW-Q, is defined by Formula 16:

$$U'(CS,\lambda) = \sum_{k=1}^{r} \frac{Q'_{max}(k,\lambda) - q'_k(CS,\lambda)}{Q'_{max}(k,\lambda) - Q'_{min}(k,\lambda)} \cdot w_k \quad (16)$$

The component services are selected according to the values obtained from SAW-Q for each abstract workflow. There are four factors that determine such value: the number of tasks the abstract model of the composite service; the number of candidates of the composite service services; quality constraints defined by the customer; and the control-flow patterns used in the abstract model. The control-flow patterns determine the execution path of a composite service. Depending on this, the component services can be invoked in sequence, in parallel, iteratively, or following alternative conditional execution paths. The quality attributes of a composite service are determined by the aggregate functions in Table **??**. The utility value ranks an implementation (of all possible) by calculating the expected behavior for each quality constraint. Therefore, since each control-flow pattern influences the utility function in various ways, the number and variety of control-flow patterns of an abstract model influence the choice of the best implementation and the best service for each task. As implementation options grow, the utility function becomes more important in the decision process to choose the best service for a task component.

9.4 Implementation and Evaluation

Implementation

Our principal hypothesis is that a dynamic composition approach based on quality restrictions that does not considers the user demand leads to implementations that can be erroneous in practice. In this section, we describe the implementation of a REST service composer based on SAW-Q, which is

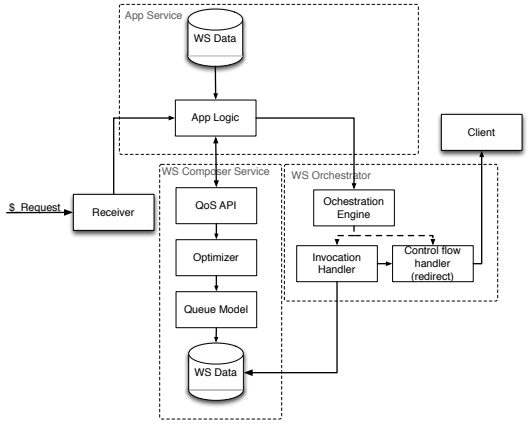


Fig. 10. Prototype architecture for dynamic composition using SAW-Q

a prototype that allows us to illustrate our results experimentally.

Figure 10 describes the prototype architecture. The module Receiver accepts the arriving requests ($\$\_Request$); it is responsible for recording service access and delivering the message to the next component. The App Logic module handles the request and determines its purpose in order to create a new instance of an existing composed service (e.g. `POST`) or to update the status of an existing one (e.g. `PUT`). When a new instance is requested, the WS Composer Service module determines the service components for the abstract process (CS_x). It accomplishes this task by invoking the Optimizer component which runs either SAW or SAW-Q in order to determine the utility values for the combination of service candidates. The Optimizer uses the Queue model library to perform equations 8, 9, and 10 (Section 9.1). The WS Composer Service defines the service invocation plan to follow. On the other hand, if the application intends to update the status of an existing composed service, then the request is derived to the WS Orchestrator module (Orchestrator Engine) that is responsible for executing the composition plan.

The Orchestrator Engine chooses at runtime the next service candidate with the highest score according the utility function determined by the WS Composer (Invocation Handles), and finally the Control-flow Handler executes the composition plan (i.e. prepares the HTTP request messages to be sent).

Control-flow patterns were implemented considering REST architectural constraints as proposed in [41]. Hence, we use redirections in order to keep application state on the client. The Control-flow Handler module extends the service response with header metadata corresponding to control-flow patterns as defined. This allows us to implement service compositions that are fully decentralized and stateless.

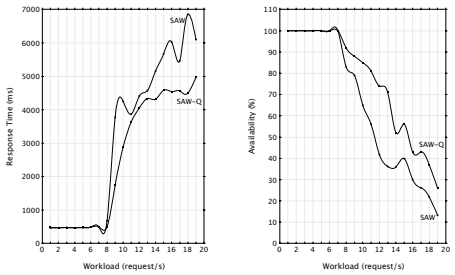
Services were implemented similarly to those of section 3.5, that is, as Python scripts (Django) running on an Apache Web server with PostgreSQL persistence. The experiment was performed online, the client run on an Intel PC with 4GB RAM, and 4 cores, the server (Ubuntu) runs on a virtual machine with 4 cores, 3GB RAM. Each service candidate was replicated up to 10 times in a LAN configuration.

Evaluation

In order to validate the obtained results empirically, we implemented both techniques SAW and SAW-Q to choose at runtime the best service candidates using the previously described prototype. Services workload (requests/second) analysis was performed using the load-testing tool JMeter, for a time long enough to obtain stable results. The results obtained in this experiment confirms that SAW-Q's rankings fit better the experimental results.

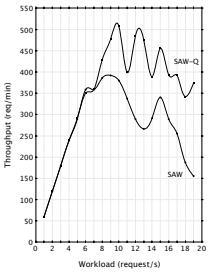
Our experimental scenario comprehends a dataset of 6 tasks corresponding to the abstract BPMN model shown in Figure 9. For each task in the business process we implemented 2 alternative services, that is, we implemented 12 services which were characterized with a random arrival rate between 4 to 10 requests per second, with random replicas (between 1 to 5) and a random capacity for each replica of 10 to 40 services in the waiting queue.

The response time is a quality attribute that negatively affects the quality of service when it grows. In this experiment the average response time for a service following SAW is higher than the one obtained using SAW-Q when the user demand is taken into account. Figure 13 shows the impact of changes in the demand from 1 to 19 requests per second on SAW and SAW-Q. For each value of the demand, we considered the average response time of the composed service with the highest score for SAW and SAW-Q (Figure 11(a)). The differences between both techniques can be appreciated when the demand scales up to 8 requests per second, from then, the services with the highest score according to



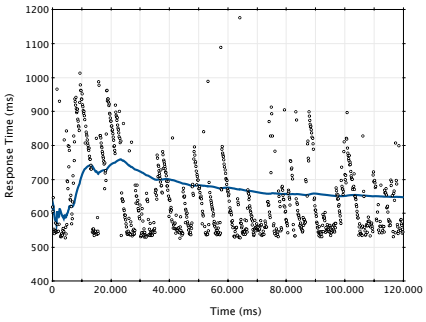
(a) Response Time according SAW and SAW-Q

(b) Availability according SAW and SAW-Q

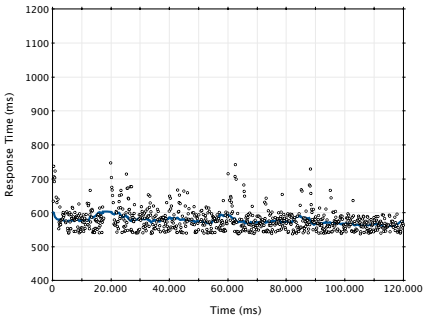


(c) Throughput according SAW and SAW-Q

Fig. 11. SAW/SAW-Q comparison.



(a) Response Time SAW



(b) Response Time SAW-Q

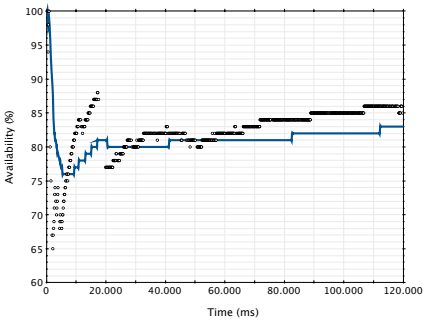
Fig. 12. SAW/SAW-Q comparison: Composed service response time.

SAW obtains a higher response time when compared to the top score services obtained by SAW-Q, hence quality of the services selected by SAW is worst than those services selected by SAW-Q.Figure 11(b) and Figure 11(c) shows the impact of SAW and SAW-Q selections on the availability and throughput of the services with highest score. Again the difference between both techniques appears since the workload is 9 requests per second.

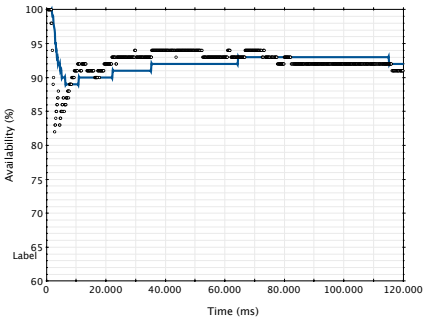
In Figure 12 we go further analyzing the situation when the demand reaches 8 requests per second. ($\lambda = 8req/s$). The dots represent the obtained results and the continuous line, the average. Note that in the composition proposed by SAW the standard deviation is bigger than the composition using SAW-Q.

The availability of the services composed using SAW-Q, on average, is greater than the compositions obtained using SAW. Figure 13 shows the availability of both services when the demand is $8req/s$. The dots are the obtained results an the continuous line shows, the measures average.

Composite service throughput using SAW-Q is slightly higher than the one using SAW which means that the SAW-Q composite service has the capacity to serve more requests per minute than the SAW composite service. Figure 14 shows the results of



(a) Availability SAW

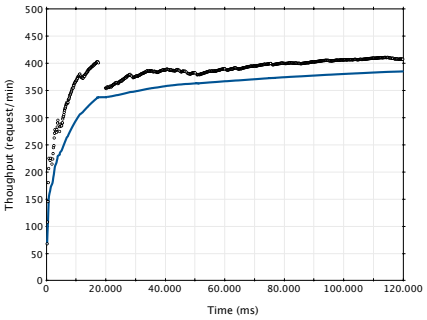


(b) Availability SAW-Q

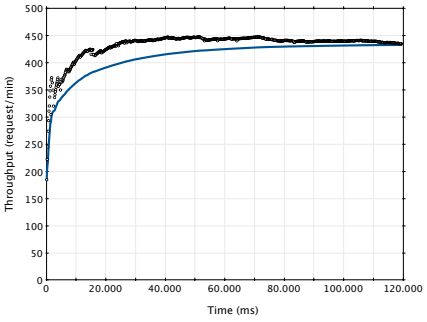
Fig. 13. Availability comparison between both techniques of composition.

both services when the demand is 8 requests per

second. Again, the dots are the obtained results and the continuous line shows the average.



(a) Throughput SAW



(b) Throughput SAW-Q

Fig. 14. Throughput comparison between SAW and SAW-Q.

10 Conclusions

10.1 Regarding Decentralized, stateless, complex service behavior in REST

In Section 7 a proposal for the design and implementation of complex composed service behavior is presented. This proposal places emphasis on REST architectural constraints, having as goal to achieve scalability and statelessness for the composed service behavior. With these considerations in mind, a set of well-known control-flow patterns that are used to implement simple and complex behavior in traditional Web services were recreated.

The main conclusion from the experience is that a decentralized, stateless implementation of a composed service satisfies REST architectural constraints and provides significant improvements regarding throughput and availability, which are non-functional goals of REST.

Second, when following REST architectural constraints and a decentralized, stateless approach where the client (User Agent) shares the interaction responsibility with the server, control-flow patterns design in REST differ from those in SOA where state (information interaction) is kept in a centralized component (the orchestrator).

Third, one of the extensibility mechanisms of HTTP, namely status codes, was used to implement the presented approach. Other alternatives could be used, such as link headers, or ad-hoc media types (e.g. a specialized JSON document). However, the precedence for link processing indicates that such messages must be processed after the representation is fully received and processed by the client and after users have performed the actions they required (e.g. click on buttons, or run javascript controls), which introduces not only delays but also security risks.

Finally, the presented approach requires that the client knows in advance how to process the messages, so that, it shall be a process-oriented User Agent. In addition, this design choice also includes the typical vulnerabilities of nowadays User Agents. Additional measures such as digital signatures must be included in order to guarantee a safe interaction between services (mediated by the User Agent).

10.2 Regarding hybrid (static and dynamic) service composition in REST

In Section 8 a technique that exploits hypermedia-centric REST service descriptions (defined at design time) is used at runtime to determine the feasibility of service composition and actually enacting a composition with an authentication service based on OAuth. Again a decentralized approach, a choreography, was followed.

The main conclusion from this approach is that hypermedia-centric REST service descriptions can actually serve as a basis for a well-behaved User Agent traverses complex paths on the Web of services. However, since such descriptions are created at design-time, dynamic changes on the service provision (e.g. changes on the service interface) could not be reflected on the descriptions. Descriptions that are out of sync with the service implementations may impede the User Agent to continue its work, although a good service description can provide information to the client developer so that the changes can be easily noticed.

Second, a QoS domain (security) is addressed in this section not only because an important authorization technique (OAuth) is an example of a

highly scalable and well-known choreography, but also because QoS-aware service composition is a field that have been extensible studied in order to support automatic and dynamic service composition. QoS attributes, particularly security are playing a major role in the current Web due to the massive scale, performance, availability and evolvability requirements that pervade modern Web applications. However, most techniques reduce QoS complexity to single values (e.g. booleans or numbers) when in practice some, such as security, shall be represented as a combination of diverse algorithms and protocols that could be available or not, or even worst shall be tried to follow in order to discover the feasibility of choosing a service as a component for a composed service.

10.3 Regarding dynamic service composition in REST

In Section 9, SAW-Q (an extension of SAW) is proposed as a novel dynamic composition technique that follows the principles of the REST style. SAW-Q models quality attributes as a function of the actual service demand instead of the traditional constant values.

The main conclusion is obtained when comparing both techniques SAW-Q is much more accurate than SAW when compared to real implementations, positively improving the quality of dynamic service compositions. Quality attributes of a REST service such as, availability, response time and throughput can be modeled with better accuracy using queuing theory since it considers implementation details that are particularly relevant for service architecture, such as the processing time of the application, the number of times the service is replicated, the maximum number of clients that can be handled by each service replica at a given time and the request that remain in the waiting queue.

Second, choosing the right candidate service in dynamic composition is a critical task. Different ways of measuring the quality of a service can lead to errors. The dynamic composition technique proposed, SAW-Q, considers the attributes of service quality as a function of the demand for requests thus obtained composed services that behave better than those determined by SAW.

Finally, the present approach contributes a deeper analysis on the scalability related attributes. Considering the request demand or workload when modeling services and composed services is critical particularly to certain quality attributes such as throughput, response time, and availability but also fault tolerance and even price, which are not considered in our study.

REFERENCES

[1] T. Erl, *Soa: principles of service design*. Prentice Hall Upper Saddle River, 2008, vol. 1.

[2] ——, *Service-oriented architecture: concepts, technology, and design*. Pearson Education India, 2005.

[3] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000. [Online]. Available: http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm

[4] T. Erl, B. Carlyle, C. Pautasso, and R. Balasubramanian, *SOA with REST: Principles, Patterns &Constraints for Building Enterprise Solutions with REST*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2012.

[5] J. Mendling and M. Hafner, "From ws-cdl choreography to bpel process orchestration," *Journal of Enterprise Information Management*, vol. 21, no. 5, pp. 525–542, 2008. [Online]. Available: http://dx.doi.org/10.1108/17410390810904274"

[6] D. Jordan, J. Evdemon, A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland *et al.*, "Web services business process execution language version 2.0," *OASIS standard*, vol. 11, p. 11, 2007.

[7] C. Pautasso, "Composing restful services with jopera," in *Software Composition*, ser. Lecture Notes in Computer Science, A. Bergel and J. Fabry, Eds. Springer Berlin Heidelberg, 2009, vol. 5634, pp. 142–159.

[8] R. Alarcón, E. Wilde, and J. Bellido, "Hypermedia-driven restful service composition," in *6th Workshop on Engineering Service-Oriented Applications (WESOA 2010)*. Springer, 2010.

[9] R. Alarcon and E. Wilde, "Linking data from restful services," in *Third Workshop on Linked Data on the Web, Raleigh, North Carolina (April 2010)*, 2010.

[10] R. Krummenacher, B. Norton, and A. Marte, "Towards linked open services and processes," in *Future Internet - FIS 2010*, ser. Lecture Notes in Computer Science, A. Berre, A. GÃşmez-PÃľrez, K. Tutschku, and D. Fensel, Eds. Springer Berlin Heidelberg, 2010, vol. 6369, pp. 68–77.

[11] S. Stadtmüller and A. Harth, "Towards data-driven programming for restful linked data," in *Workshop on Programming the Semantic Web (ISWCâĂŹ12)*, 2012.

[12] R. Verborgh, T. Steiner, D. Deursen, R. Van de Walle, and J. Valles, "Efficient runtime service discovery and consumption with hyperlinked restdesc," in *Next Generation Web Services Practices (NWeSP), 2011 7th International Conference on*, oct. 2011, pp. 373 –379.

[13] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto, "Web services choreography description language version 1.0," *W3C candidate recommendation*, vol. 9, 2005.

[14] G. Decker, "Process choreographies in service-oriented environments," Ph.D. dissertation, Masters thesis, Hasso Plattner Institute at University of Potsdam, 2006.

[15] M. zur Muehlen, J. V. Nickerson, and K. D. Swenson, "Developing web services choreography standardsâĂŤthe case of {REST} vs. {SOAP}," *Decision Support Systems*, vol. 40, no. 1, pp. 9 – 29, 2005, web services and process management. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167923604000612

[16] C. Pautasso, "Restful web service composition with bpel for rest," *Data Knowl. Eng.*, vol. 68, no. 9, pp. 851–866, September 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1550965.1551240

[17] O. Nierstrasz and T. D. Meijler, "Requirements for a composition language," *Lecture Notes in Computer Science*, vol. 924, pp. 147–161, 1995.

[18] C. Pautasso and G. Alonso, "The {JOpera} visual composition language," *Journal of Visual Languages and Computing*, vol. 16, no. 1âĂŞ2, pp. 119 – 152, 2005, 2003 {IEEE} Symposium on Human Centric Computing Languages and Environments. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1045926X04000400

[19] F. Rosenberg, F. Curbera, M. Duftler, and R. Khalaf, "Composing restful services and collaborative workflows: A lightweight approach," *Internet Computing, IEEE*, vol. 12, no. 5, pp. 24–31, Sept 2008.

[20] G. Decker, A. Lüders, H. Overdick, K. Schlichting, and M. Weske, "Restful petri net execution," in *WS-FM*, 2008, pp. 73–87.

[21] X. Xu, L. Zhu, Y. Liu, and M. Staples, "Resource-oriented architecture for business processes," in *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, Dec 2008, pp. 395–402.

[22] H. Zhao and P. Doshi, "Towards automated restful web service composition," in *Web Services, 2009. ICWS 2009. IEEE International Conference on*, July 2009, pp. 189–196.

[23] D. Menasce, "Qos issues in web services," *Internet Computing, IEEE*, vol. 6, no. 6, pp. 72 – 75, nov/dec 2002.

[24] R. Kübert, G. Katsaros, and T. Wang, "A restful implementation of the ws-agreement specification," in *Proceedings of the Second International Workshop on RESTful Design*, ser. WS-REST '11. New York, NY, USA: ACM, 2011, pp. 67–72. [Online]. Available: http://doi.acm.org/10.1145/1967428.1967444

[25] S. Graf, V. Zholudev, L. Lewandowski, and M. Waldvogel, "Hecate, managing authorization with restful xml," in *Proceedings of the Second International Workshop on RESTful Design*, ser. WS-REST '11. New York, NY, USA: ACM, 2011, pp. 51–58. [Online]. Available: http://doi.acm.org/10.1145/1967428.1967442

[26] J. P. Field, S. G. Graham, and T. Maguire, "A framework for obligation fulfillment in rest services," in *Proceedings of the Second International Workshop on RESTful Design*, ser. WS-REST '11. New York, NY, USA: ACM, 2011, pp. 59–66. [Online]. Available: http://doi.acm.org/10.1145/1967428.1967443

[27] D. Allam, "A unified formal model for service oriented architecture to enforce security contracts," in *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development Companion*, ser. AOSD Companion '12. New York, NY, USA: ACM, 2012, pp. 9–10. [Online]. Available: http://doi.acm.org/10.1145/2162110.2162120

[28] J. Hongbin, Z. Fengyu, and X. Tao, "Security policy configuration analysis for web services on heterogeneous platforms," *Physics Procedia*, vol. 24, Part B, no. 0, pp. 1422 – 1430, 2012, international Conference on Applied Physics and Industrial Engineering 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1875389212002544

[29] F. Rosenberg, F. Curbera, M. J. Duftler, and R. Khalaf, "Composing RESTful services and collaborative workflows: A lightweight approach," *IEEE Internet Computing*, vol. 12, no. 5, pp. 24–31, 2008. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MIC.2008.98

[30] D. DâĂŹMello, V. Ananthanarayana, and S. Salian, "A review of dynamic web service composition techniques," in *Advanced Computing*, ser. Communications in Computer and Information Science, N. Meghanathan, B. Kaushik, and D. Nagamalai, Eds. Springer Berlin Heidelberg, 2011, vol. 133, pp. 85–97.

[31] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.

[32] N. Russell, Arthur, W. M. P. van der Aalst, and N. Mulyar, "Workflow control-flow patterns a revised view," BPMcenter.org, New York, NY, USA, Tech. Rep. BPM-06-22, 2006.

[33] G. D. Ivan Zuzak, Ivan Budiselic, "A finite-state machine approach for modeling and analyzing restful systems," *Web Engineering*, vol. 10, no. 4, pp. 353–390, 2011.

[34] M. Maleshkova, C. Pedrinaci, J. Domingue, G. Alvaro, and I. Martinez, "Using semantics for automating the authentication of web apis," in *The Semantic Web - ISWC 2010*, ser. Lecture Notes in Computer Science, P. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J. Pan, I. Horrocks, and B. Glimm, Eds. Springer Berlin / Heidelberg, 2010, vol. 6496, pp. 534–549.

[35] E. Hammer-Lahav, "The oauth 1.0 protocol," 2010.

[36] Q. Tao, H. you Chang, C. qin Gu, and Y. Yi, "A novel prediction approach for trustworthy qos of web services," *Expert Systems with Applications*, vol. 39, no. 3, pp. 3676 – 3681, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417411013698

[37] H. A. Taha, *Operations research: an introduction*. Pearson/Prentice Hall, 2007.

[38] M. Alrifai, T. Risse, and W. Nejdl, "A hybrid approach for efficient web service composition with end-to-end qoS constraints," *TWEB*, vol. 6, no. 2, p. 7, 2012. [Online]. Available: http://doi.acm.org/10.1145/2180861.2180864

[39] R. Hamadi and B. Benatallah, "A petri net-based model for web service composition," in *Fourteenth Australasian Database Conference (ADC2003)*, ser. CRPIT, K.-D. Schewe and X. Zhou, Eds., vol. 17. Adelaide, Australia: ACS, 2003, pp. 191–200. [Online]. Available: "http://crpit.com/confpapers/CRPITV17Hamadi.pdf"

[40] M. Zeleny and J. L. Cochrane, *Multiple criteria decision making*. McGraw-Hill New York, 1982, vol. 25.

[41] J. Bellido, R. Alarcón, and C. Pautasso, "Control-flow patterns for decentralized restful service composition," *ACM Trans. Web*, vol. 8, no. 1, pp. 5:1–5:30, December 2013. [Online]. Available: http://doi.acm.org/10.1145/2535911