The Knowledge Engineering Review, Vol. 29:2, 265–280. © Cambridge University Press, 2014
doi:10.1017/S0269888914000095

A goal-based approach to engineering capacity-driven Web services

ZAKARIA MAAMAR[1], SAMIR TATA[2], KOKOU YETONGNON[3], DJAMAL BENSLIMANE[4] and PHILIPPE THIRAN[5]

[1]*Zayed University, PO Box 19282, Dubai, UAE;*
e-mail: zakaria.maamar@zu.ac.ae;
[2]*Computer Science Department, Telecom SudParis, 9 Rue Charles Fourier, 91011 Evry Cedex, France;*
e-mail: samir.tata@it-sudparis.eu;
[3]*Laboratoire Le2i - UMR 5158UFR Sciences et Techniques BP 47870, 21078 Dijon Cedex, France;*
e-mail: kokou@u-bourgogne.fr;
[4]*IUT - Université Claude Bernard Lyon 143 Bd du 11 Novembre 191869622, Villeurbanne Cedex, France;*
e-mail: djamal.benslimane@liris.cnrs.fr;
[5]*Faculty of Computer Science, University of Namur, 21 Rue Grandgagnage, 5000 Namur, Belgium;*
e-mail: pthiran@fundp.ac.be

Abstract

This paper discusses a goal-based approach for the engineering of capacity-driven Web services. In this approach, goals are established to first, define the roles that these Web services will play in implementing business applications, second, frame the requirements that will be put on these Web services, and third, identify the processes in terms of business logics that these Web services will carry out. Because of the nature of capacity-driven Web services compared with regular (i.e. mono-capacity) Web services, their engineering in terms of design, development, and deployment takes place in a different way. A Web service that is empowered with several capacities, which are basically separate groups of operations to execute, has to choose one capacity for triggering at run-time. To this end, the Web service takes into account different types of requirements like data and privacy that are put on each capacity that empowers this Web service.

1 Introduction

In Maamar *et al.* (2009), we discuss the notion of *capacity* and how it weaves smoothly into Web services. By capacity, we mean how a Web service is empowered with several sets of operations from which it selectively triggers a particular set with respect to some run-time environment requirements. The description of these sets of operations is included in the *C*apacity-driven Web Services Description Language (*C*-WSDL) document of the capacity-driven Web service. In this paper, we continue our discussions on capacity-driven Web services with focus this time on how *goals* help in engineering such Web services upon which enterprise applications can be built. By goal, we mean a high-level statement that (i) channels the roles that a Web service will play in a business application, (ii) frames the requirements that will be posed on this Web service, and (iii) sets the business scenarios that this Web service will implement. The rationale of examining capacities is backed by the statement that '*most Web services platforms are based on a best-effort model, which treats all requests uniformly, without any type of service* ***differentiation*** *or* ***prioritization***' (Malay Chatterjee *et al.*, 2005). We refer to the Web services reported in this statement as mono-capacity. Currently, Web services process users' requests without considering, for example, if users are outside or inside, which definitely requires special attention to the bandwidth to use.

https://doi.org/10.1017/S0269888914000095 Published online by Cambridge University Press

Web services are gaining momentum in academia and industry by achieving the promise of developing loosely coupled, cross-enterprise business applications. To sustain this momentum, we stressed several times the importance of designing and developing Web services along the flexibility, stability, and autonomy requirements[1] (Maamar *et al.*, 2006). By flexibility, we refer to a Web service that can adapt itself so that it accommodates the characteristics of the business scenario it implements. By stability, we refer to a Web service that can resist to unforeseen changes so that it maintains operation continuity and recovers to normal levels of operation after disturbances. Finally, by autonomy, we refer to a Web service that can evaluate the possible rewards it is eligible for so that it either accepts or rejects processing customers' requests. A customer could be a user or another Web service.

Capacity is an intrinsic element of the approach we proposed in response to the challenges that emerge from the aforementioned requirements (Maamar *et al.*, 2009). In this approach, we let Web services assess themselves before they accept to participate in any composition scenario, and assess the requirements and track the changes that are posed by and made in the execution environment, respectively, before these Web services select a certain capacity to trigger. On the one hand, requirements could be related to network reliability, security measures, and data quality. On the other hand, changes could be related to drop in network bandwidth, peer appearance without prior notice, and computing resources mobility and sometimes failure.

The deployment of capacity-driven Web services stresses out the importance of assisting those who are put in the front line of designing, developing, and maintaining such Web services. There is a lack of guidelines that would offer the necessary assistance to service engineers. Part of this assistance is usually known as requirement engineering in the software field. Our contributions in this paper are manifold: set the goals that drive the engineering of capacity-driven Web services; provide guidelines to identify capacities of Web services; identify the requirements that the environment poses on Web services so that these ones satisfy these requirements through capacities; and propose techniques and notations to describe both capacities and requirements.

Though the notion of capacity is not explicitly defined in the Web services literature, we came cross some terms like capability and ability. Arroyo *et al.* discuss Web services' capabilities and constraints as part of the activities of the Web Service Modeling Ontology (WSMO) working group (Arroyo *et al.*, 2004). The authors recognize that addressing clients' and Web services' capabilities and constraints is a prerequisite to the success of Business-to-Customer (B2C) and Business-to-Business (B2B) transactions. In the same vein, Birman notes that the lack of mechanisms to model Web services' capabilities and constraints prevents Web services from taking over complex case studies that require more than simple request/response interaction patterns, and thus confines Web services to simple case studies (Birman, 2004). In Vukovic and Robinson (2004), Vukovic and Robinson's adaptive planning approach to compose Web services highlights the importance of selecting the appropriate actions, means, and resources based on the current requirements of the environment. The mail replication system that is used as a running scenario indicates the suitable procedure to adopt based on user location, activity, computing device, and network bandwidth. Examples of procedures consist of displaying mail headers when the user's activity and location are walking and street, respectively, and displaying full mails when the users' activity and location are working and at desk, respectively. Each procedure calls for a specific set of operations, which are in line with the way we envision the use of capacities.

In this paper, we present a goal-based approach to engineering capacity-driven Web services. In this approach, the goals 'kick-off' this engineering in terms of requirements to identify, capacities to develop, and business logic to adapt. This paper is organized as follows. Section 1 provides an overview of capacity-driven Web services and motivates the use of goals to engineer these Web services. Section 2 suggests a background material on some topics discussed in this paper. Section 3 introduces our goal-based approach to engineering capacity-driven Web services. This engineering

[1] These requirements make Web services sensible to context.

https://doi.org/10.1017/S0269888914000095 Published online by Cambridge University Press

happens along three perspectives denoted by requirement, capacity, and business logic. Prior to concluding in Section 5, Section 4 briefly summarizes how capacity-driven Web services are put into action following their engineering.

2 Background

2.1 Some definitions

Web service is '*a software application identified by a URI, whose interfaces and binding are capable of being defined, described, and discovered by XML artifacts, and supports direct interactions with other software applications using XML-based messages via Internet-based applications*' (W3C). A Web service has a functionality that shows the type of 'service' it offers to users and peers as well. `currencyConversion` and `carRental` are examples of functionalities.

Composition targets users' requests that cannot be satisfied by any single, available Web service, whereas a composite Web service obtained by combining available Web services may be used. A good number of languages to compose Web services exist, for instance the Web Services Business Process Execution Language (BPEL, *defacto* standard) and the Web Services Choreography Description Language (WSCDL).

Capacity corresponds to a set of actions to carry out in response to invoking the functionality of a Web service at run-time (Maamar *et al*., 2009). A functionality differentiates a Web service from other peers, although it is common that independent bodies develop Web services with similar functionalities but different non-functional properties. Concretely speaking, the actions in a capacity (i) correspond to operations in a WSDL document and (ii) vary according to the business application domain. Some actions in *PlaceBookingWS* are `checkPlaceAvailability` and `confirmBooking`. Several capacities with their respective actions satisfy the unique functionality of a Web service in different ways: which capacity to select and activate out of the available capacities requires assessing the environment so that relevant details about this environment are collected and prepared for this selection. According to the time of the day, *GuestWS* contacts invitees by either making phone calls in the morning or sending emails in the evening.

Environment assessment identifies the requirements that a Web service needs to satisfy before this Web service first, accepts to participate in a new composition scenario (on top of participating in other ongoing compositions) and second, selects a capacity to trigger after accepting this participation. As a result, environment assessment identifies two types of requirements. Those associated with composition scenarios and those associated with capacity activation. Example of a composition requirement is the maximum number of compositions that a Web service can take part in at a time. And example of a capacity requirement is the minimum network bandwidth to maintain so that a Web service guarantees regular live-content delivery to users of handheld mobile devices.

2.2 Running example

Our running example concerns a university student who organizes a cookout party for his recent graduation. We, hereafter, suggest some Web services that implement this party.

CateringWS searches for and contacts catering companies according to criteria like budget allocated, number of guests expected, and type of cuisine.

GuestWS sends friends invitations, keeps track of invitations confirmed, and reminds guests in case of late confirmations.

PlaceBookingWS looks for a place to host the party, books the place, and completes the necessary paperwork like making down payment to confirm the booking.

WeatherWS checks weather forecast for the day planned of the party. In case of bad weather, the party takes place at the university's function facility. In this paper, *WeatherWS* illustrates some aspects of our engineering approach. Briefly, *WeatherWS* accesses a remote database to

https://doi.org/10.1017/S0269888914000095 Published online by Cambridge University Press

check out the weather forecast of a city on a certain day and sends users reports. It might happen that the access to the database fails due to network problems or database unavailability.

2.3 Related work on Web services engineering

Our literature review identified a good number of research initiatives that study Web services engineering (Kirda *et al.*, 2003; Robinson, 2003; Chris Gibson, 2004; van Eck & Wieringa, 2004; Jha, 2006; Tsai *et al.*, 2007). Unfortunately, none of these initiatives embraces goals as a driving element of the efforts to put into this engineering. Furthermore, our literature review included other initiatives on goal-driven requirement engineering (Donzelli, 2004; Damas *et al.*, 2006; Elghazi, 2007).

In Chris Gibson (2004), Chris Gibson stresses the importance of developing a requirement specification for software applications built around Web services. The author reports that deficiencies in software requirements are the leading cause of failure in software projects. The use of Web services to provide B2B online solutions turns out useful; Web services simplify application development and reduce development risk and cost. The requirement engineering process of Chris Gibson is commonly adopted by the IT community with some well-known steps such as elicitation, analysis, modeling and specification, and verification.

In Jha (2006), Jha develops an approach based on problem frames for Web services' requirements. The approach aligns initiatives on Web services with the strategies of the enterprise so that the objectives, needs, and context of this enterprise are captured. Jha adopts the definition of Zave and Jackson that requirements are all about describing a client's problem domain, determining what desired effects the client wants to exert upon that domain, and specifying the external face of the proposed systems to enable those desired effects to occur and to give designers a specification that guides them in building such systems (Zave & Jackson, 1997). To capture an enterprise's objective, needs, and context, Jha suggests two steps: understand the enterprise's business strategy and overall objective, and use progression of problems to describe this enterprise's business objective and the business context from strategy to implementation.

In Kirda *et al.* (2003), Kirda *et al.* note that the problem of supporting the automatic integration of Web services into Web sites has received little attention from the research community. To remedy this problem, the authors describe how Web services can be modeled, implemented, composed, and automatically integrated into Web sites using the Device-Independent Web Engineering (DIWE) framework. DIWE promotes the separation between three layers known as layout, content, and application logic.

In Robinson (2003), Robinson focusses on monitoring the requirements of Web services as part of the process of making software applications reliable. Businesses that embrace Web services as a development technology are automatically vulnerable to the direct shortcomings of these Web services. Addressing these shortcomings before they arise calls for a preventive strategy that Robinson summarizes with the following actions: discover obstacles, assess the derivation of assigned monitors from obstacles, and derive Web service monitors from high-level requirement descriptions. Interesting here is the definition of obstacle, which is an environment condition that prevents requirement satisfaction. Sudden drop in network bandwidth cancels content delivery, unless other alternatives are planned.

In Tsai *et al.* (2007), Tsai *et al.* discuss Service-Oriented System Engineering (SOSE) with focus on Service-Oriented Requirement Engineering (SORE). SORE is different from other traditional requirement engineering approaches since the concerned applications have to comply with the general guidelines of service-oriented architecture. Tsai *et al.* indicate the following characteristics of SORE: reusability-oriented and cumulative, domain-specific, framework-oriented analysis, model-driven development, evaluation-based, user-centric analysis and specification, and finally, policy-based computing.

In Agre *et al.* (2006), Agre *et al.* discuss the engineering of semantic Web services as part of the IST FP6 INFRAWEB project. The use of such Web services is still limited due to different reasons

https://doi.org/10.1017/S0269888914000095 Published online by Cambridge University Press

such as complexity of both OWL-S and WSMO. The objective of this project is to develop a semantic service engineering framework to enable the creation, maintenance, and execution of WSMO-based semantic Web services. The potential users of this framework include semantic Web services providers, semantic Web services brokers, semantic Web services application providers, and semantic Web services consumers. In Aoyama *et al.* (2002), Aoyama *et al.* discuss the promises and challenges of Web services engineering. This engineering covers every aspect of a Web service from development, deployment, use, to evolution along with analysis, architectures, development methodologies, descriptions, testing, development environments, management, and applications. Platform, cross-organization boundaries, and business models are among the challenges that Aoyama *et al.* report.

Last but not least, in van Eck and Wieringa (2004), van Eck and Wieringa examine the requirements of a specific type of Web services. This specific type of Web services supports other services that, in fact, are not Web services. van Eck and Wieringa use the wording of product experience augmenters to qualify such Web services. In addition, they claim that most Web services in the future will augment existing products or services, rather than constituting an independent economic offering. Some characteristics that feature product-experience-augmenters Web services include: their functional maintenance is the responsibility of the marketing and sales department or the departments that offer the primary product, and they need to properly fit into the business processes.

Although the aforementioned paragraphs offer a reasonable snapshot of the field of Web services engineering, there is no clear vision that articulates how goals could smoothen this engineering. Questions like how to identify and associate goals with Web services, how to review Web services design because of changes in goals, or how to put requirements on Web services are left unanswered and thus, responses are provided on a case-by-case basis.

Prior to concluding this literature review, we mention that capacity-driven Web services fit perfectly into the research stream of non-functional requirements in software engineering (Chung *et al.*, 1999). By letting Web services select the appropriate operations (in fact, capacities) to execute in response to some requirements, this would allow maintaining a certain level of satisfaction of these non-functional requirements. Different works on service-level agreements in relation to non-functional requirements exist such as Heiko *et al.* (2006) and Oldham *et al.* (2006). These agreements contain clauses on how services are delivered according to different factors like computing resource availabilities, users' preferences, and network bandwidth. This type of delivery requires mechanisms that guarantee the satisfaction of these clauses at run time. These mechanisms can be implemented through capacities that, as we propose in this paper, are activated after constraint satisfaction. Each time a clause in an agreement needs to be guaranteed, a dedicated capacity is selected. Another initiative that could help specify some elements of capacity-driven Web services is TROPOS (Kazhamiakin *et al.*, 2004). Kazhamiakin *et al.* discuss a framework that combines business processes and business requirements. This framework describes how an organizational strategy is operationalized into activities and implemented by business processes. For business requirement modeling, Kazhamiakin *et al.* adopt TROPOS, which is a goal-driven, agent-oriented software methodology that covers the complete development cycle of a system. TROPOS can be used to define business requirements and identify them with the corresponding business processes. Applied to capacity-driven Web services, TROPOS could be used to model the requirements to put on these Web services as well as the goals that drive the engineering of these Web services.

3 Our goal-based engineering approach

Prior to delving into the details of our goal-based engineering approach, we provide a definition of what a goal can stand for. According to van Lamsweerde (2001), '*a goal is an objective the system under consideration should achieve. Goals may be formulated at different levels of abstraction, ranging from high-level, strategic concerns to low-level, technical concerns*'.

https://doi.org/10.1017/S0269888914000095 Published online by Cambridge University Press

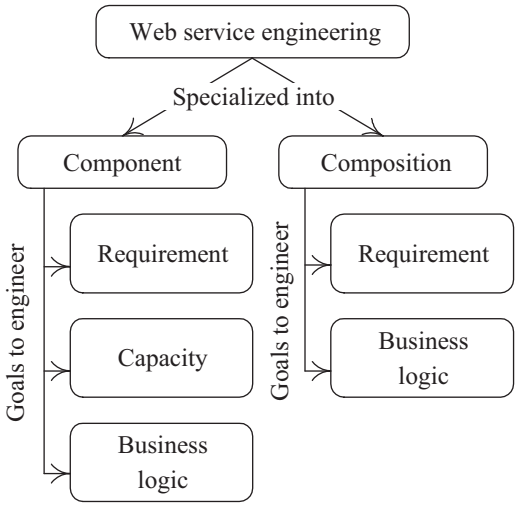


Figure 1 Conceptual representation of the goal-based engineering approach

3.1 Overview

Figure 1 illustrates our goal-based engineering approach for (component) capacity-driven Web services. This approach can be extended to 'composite' Web services at a later stage but this is outside this paper's scope. Basically, our approach sets the goals that would 'kick-off' the exercise of engineering a capacity-driven Web service in terms of requirements, capacities, and business logic. The first goal identifies the types of requirements that affect the way the Web service accomplishes its functionality (Section 3.2). The second goal details the capacities that empower the Web service so that this Web service satisfies these requirements (Section 3.3). Finally, the third goal reviews the business logic that underpins the functionality of the Web service following the empowerment of this Web service with capacities (Section 3.4).

Through our engineering approach for capacity-driven Web services, the three types of goals we suggest can address the following questions:

Q_0: What requirements could be posed on a Web service?
To answer Q_0, the goals characterize the requirements to put on (i) the data manipulated by a Web service, (ii) the communications means used by a Web service, and (iii) the computing resources upon which a Web service performs (Section 3.2).

Q_1: What actions per capacity need to be developed in order to satisfy the requirements defined in Q_0?
To answer Q_1, the goals characterize how to structure capacities in a Web service in terms of actions and input and output arguments (Section 3.3).

Q_2: How the business logic of a Web service needs to be adapted because of the capacities defined in Q_1?
To answer Q_2, the goals characterize the changes that are made in a business logic following the weaving of capacities into this business logic (Section 3.4).

3.2 Requirement engineering through goals

In this first step of the engineering approach, the goal analysis contributes towards (i) reflecting the dynamic nature of the environment, (ii) understanding the types of requirements that this environment poses on Web services, and (iii) linking some of these requirements to capacity development. By dynamic nature, we mean Web services appearing and disappearing without prior notice, Web services resuming and suspending operation, sudden drop in network bandwidth, etc. As per the definition of 'environment assessment' in Section 2.1, we identify two **goals** and associate them with 'composition requirements' and 'capacity requirements', respectively.

https://doi.org/10.1017/S0269888914000095 Published online by Cambridge University Press

The former goal regulates the participation of a Web service in a new composition scenario, and the latter goal regulates the activation of at least one capacity in a Web service at run-time.

3.2.1 Composition requirements

In Maamar *et al.* (2006), we introduced the Web services instantiation principle to illustrate the conditional participation of a Web service in composition scenarios. Upon acceptance, this participation happens through a Web service instance. In compliance with this instantiation principle, a Web service is looked at from three dimensions: past compositions, current compositions, and forthcoming compositions. Requirements apply to forthcoming compositions and thus, can limit the participation of a Web service in these compositions.

DEFINITION 1 **(Composition Requirement).** The composition requirement on a Web service is a tuple $COR_{WS} = (arg_i, val_i, desc_i, Ont)$ where:

- arg_i is the name of a type of composition requirement.
- val_i is a value (numerical, string, etc.) assigned to arg_i.
- $desc_i$ is a narrative description of the composition requirement.
- Ont refers to the ontology defining arg_i. □

The set of all possible composition requirements is denoted by COR.

Examples: *The following elements populate* COR:

- arg_1*: maximum number of participations,* val_1*: 10,* $desc_1$*: the maximum number of participation for a Web service in compositions at a time is limited to 10, Ont:* $ONT_{comp-req}$.
- arg_2*: date of maintenance,* val_2*: every Wednesday of the week,* $desc_2$*: on Wednesdays, a Web service does not take part in any composition because of maintenance (upgrade, code change, etc.), Ont:* $ONT_{comp-req}$.

3.2.2 Capacity requirements

A Web service implements its functionality through capacities: which capacity to activate at run-time depends on the nature of requirements that are put on this Web service. The requirements apply to all Web services but are fine tuned depending of the nature of a Web service like application domain. We classify requirements on capacities into different types with focus in this paper on the following three types[2]:

1. Data requirement is about the quality of data that a Web service receives, manipulates, and probably sends out. This requirement is about freshness (when were the data obtained), source (who is the sender of the data), and validity (when do the data expire) of data.
2. Network requirement is about the nature of communication means that a Web service uses to interact with users and peers. This requirement is about bandwidth, throughput, and reliability.
3. Resource requirement is about the computing facilities upon which the performance of a Web service is scheduled. This requirement is about availability and reliability.

DEFINITION 2 **(Capacity Requirement).** The capacity requirement on a Web service is a tuple $CAR_{WS} = (arg_i, val_i, desc_i, thre_i, Ont)$ where:

- arg_i is the name of a capacity requirement.
- val_i is a value (numerical, string, etc.) assigned to arg_i.

[2] Additional requirements can be added such as security. This requirement is about the protection measures that are put in place such as encryption protocol and length of private keys.

https://doi.org/10.1017/S0269888914000095 Published online by Cambridge University Press

- $desc_i$ is a narrative description of the capacity requirement.
- $thre_i$ is a threshold value that keeps the capacity requirement illustrated with arg_i satisfied despite changes in the environment that could affect val_i.
- *Ont* refers to the ontology defining arg_i. □

The set of all possible capacity requirements is denoted by CAR.

Examples: *The following elements populate* CAR:

- arg_1*: data freshness,* val_1*: current,* $desc_1$*: all data to be used by a Web service need to be recent, that is, as of today,* $thre_1$*: null, Ont:* $ONT_{cap-req}$.
- arg_2*: resource availability,* val_2*: 80%,* $desc_2$*: resources upon which a Web service runs, need to have an 80% availability level,* $thre_2$*: 5% decrease in resource availability is acceptable to the Web service and the current activated capacity can continue to be used, Ont:* $ONT_{cap-req}$.

3.3 Capacity engineering through goals

In this second step of the engineering approach, the goal analysis contributes towards (i) understanding how to structure capacities, (ii) assessing how capacities are triggered, and last but not least (iii) linking these capacities to requirements. As per the definition of 'capacity' in Section 2.1, we identify **goals** with respect to the capacities that empower a Web service. A capacity is a set of operations that a Web service carries out upon receiving requests from users or peers. A Web service could have several capacities that are differently structured according to the functionality it implements and the requirements that these capacities satisfy.

DEFINITION 3 **(Web Service Capacity).** The capacity in a Web service is a couple $CAP_{WS} = (OP, access)$ where:

- OP is a set of operations. An operation $Op_i \in OP$ consists of a name N, a set of input arguments *Input* along with their data types, and a set of output arguments *Output* along with their data types, and is written as (N, *Input*, *Output*).
- *access* is the access modifier of a capacity whether *public* or *private*. □

The set of all defined capacities is denoted by CAP. The rationale of public and private access modifiers is given later.

Example: *Let us consider* $CAP^1_{PlaceBookingWS}$ *of PlaceBookingWS. It is specified as follows:* $(\{Op_1, Op_2\}, public)$ *where* $Op_1 = (checkplace, in(date{:}Date), out(availability{:}Boolean))$, $Op_2 = (confirmplace, in(date{:}Date), out(confirmation{:}Boolean))$, *and access is public.*

The selection of a capacity in a Web service WS for triggering is subject to satisfying some capacity requirements, that is, $CAR^{i=1,\ldots,n}_{WS}$. As a result, we define a function that associates a capacity with the capacity requirements as per Definition 4.

DEFINITION 4 **(Association Function).** Let CAP and CAR be, respectively, the set of all capacities defined in a Web service and the set of all possible capacity requirements on Web services. The association function is defined as follows: $Assoc : CAP_{WS} \rightarrow 2^{CAR}$. □

The association function *Assoc* connects each capacity in a Web service WS with a set (possibly empty) of capacity requirements (2^{CAR} is the power set of CAR). This connection is done manually, that is, designer driven.

Example: *Let us continue using* $CAP^1_{PlaceBookingWS}$. $Assoc\ (CAP^1_{PlaceBookingWS}) = \{CAR^1_{PlaceBookingWS}\}$ *where* $CAR^1_{PlaceBookingWS}$ *is defined as follows:* $< datafreshness, current, \cdots, null, Ont_{req-cap} >$.

https://doi.org/10.1017/S0269888914000095 Published online by Cambridge University Press

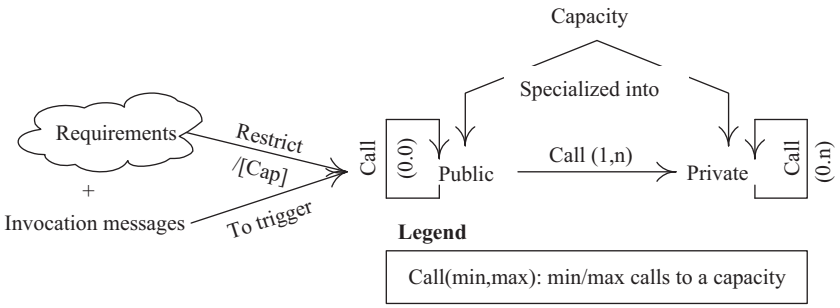


Figure 2 Capacity categorization

Based on Section 3.3, the following comments are made:

- A capacity in WS, which does not have any requirement to satisfy is referred to as *default*, that is, $Assoc(CAP^{i}_{WS}) = \emptyset$. A default capacity is automatically selected when none of the available capacities in WS satisfies the current capacity requirements. By having a default capacity, WS can always guarantee the satisfaction of a user's request.
- Independent capacities in WS can have common operations, that is, $\cap_{i=1}^{n} CAP^{i}_{WS} \neq \emptyset$.
- Common operations in independent capacities can be overloaded if necessary. For example, Op_1 takes one input argument in CAP^{1}_{WS}, for example, in(username:String), and two input arguments in CAP^{n}_{WS}, for example, in(username:String, password:String).

In Section 3.3, a capacity has either public or private access modifier like in object-oriented programming languages (Figure 2)[3]. On the one hand, public capacities are offered to the external environment namely users and other Web services. Submitting invocation messages to public capacities requires fulfilling capacity requirements. On the other hand, private capacities are hidden as their name hints and are just called by public capacities. By doing so, the privacy of a Web service in terms of offered and existing capacities is maintained; only the necessary capacities are exposed. It should be noted that private capacities might call each other if necessary ((0, n) cardinality in Figure 2), but this is not the case with public capacities ((0, 0) cardinality in Figure 2) since they all implement the unique functionality of a Web service.

3.4 Business logic engineering through goals

In this third step of the engineering approach, the goal analysis contributes towards (i) understanding the current practices in terms of business processes, (ii) motivating the need of changing these processes, and last but not least (iii) linking these process changes to requirement satisfaction.

As per the definition of 'Web service functionality' in Section 2.1, we identify a **goal** with respect to the functionality (e.g. `currencyConversion`) of a Web service. A functionality is about a business logic that describes the processes to execute in terms of how, when, and where. A business logic is domain-application dependent (e.g. education, tourism) and varies from one case study to another according to elements such as users (e.g. minimum age to submit an application), security (e.g. maximum length of encrypted key), and legal (e.g. minimum VAT rate).

DEFINITION 5 **(Web Service Goal).** The goal of a Web service is a couple $G_{WS} = (fct, bl)$ where:

- fct is a narrative description of the functionality.
- bl is a specification of the business logic.

[3] In Figure 2, *Restrict/[Cap]* means that requirements are here to restrict the capacity that a Web service can select. And, *Call(min,max)* means the minimum and maximum times that a capacity can be called by another capacity; for example, *Call(0,0)* means that a capacity is never called by any other capacity; this applies to public capacities, only. Contrarily, private capacities could be either called or never called *Call(0,n)*.

https://doi.org/10.1017/S0269888914000095 Published online by Cambridge University Press

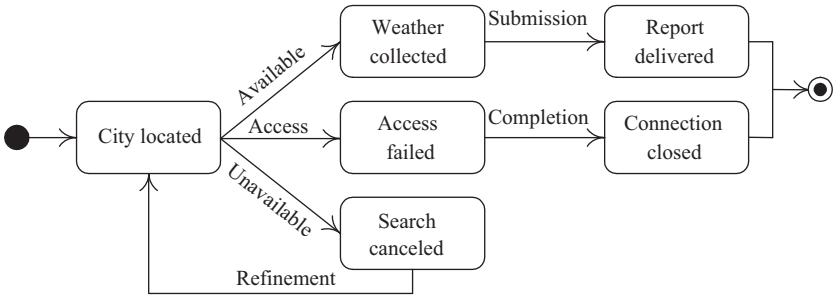


Figure 3 Business logic of *WeatherWS*

The specification of a business logic uses *state charts* but other techniques like *petri-nets* could be used.

DEFINITION 6 **(Web Service Business-Logic).** The *S*tate *C*hart SC of the business logic of a Web service WS is a tuple $SC_{WS(bl)} = (S, S^F, L, T, s^0)$ where:

- S is a finite set of state names.
- $s^0 \in S$ is the initial state in SC_{WS}.
- $S^F \subset S$ is a finite set of final states.
- L is a set of labels. In state chart, a label consists of an event component E, a condition component C, and an action component A, and is written as $E[C]/A$. For representation purposes in Figure 3, we just name labels without giving full details of the conditions and actions. In our goal-based engineering approach, condition C is split into pre-condition represented as $\bullet C$ and post-condition represented as $C\bullet$. If state s_i is directly connected to state s_{i+1}, the post-conditions of s_i subsume the pre-conditions of s_{i+1}, that is, $s_i(C\bullet) \Rightarrow s_{i+1}(\bullet C)$.
- $T \subseteq (S \times L \times S)$ is a finite set of transitions. Each transition $t \in T, t = (s^{src}, l, s^{tgt})$ consists of a source state $s^{src} \in S$, a target state $s^{tgt} \in S$, and a transition label $l \in L$. □

Example: *Figure 3 is a state chart that represents* `weatherForecast` functionality of WeatherWS. Several states like `city-located`, `report-delivered`, and `search-canceled` are included in this state chart. Moreover, this state chart includes transitions such as (`city-located`, `unavailable`, `search-canceled`) where `city-located` and `search-canceled` are the source and target states, respectively, and `unavailable` is the label of the transition.

The business logic in Figure 3 is an example of how a 'regular' (i.e. mono-capacity) Web service (*WeatherWS*) functions independently of the requirements of type capacity (can be security, network, trust, etc.) that are posed on this Web service, and thus can restrict the functioning of this Web service. Unfortunately, the specification of a business logic is tightened to the functionality to offer and does not show how these capacity requirements are handled nor how this functionality is affected because of these requirements. To address this lack of handling, we review the business logic of a Web service with focus on the needs of each type of capacity requirement. As a result, we define a conversion function that

- takes two inputs namely a business logic *bl* (represented here with a state chart) and a set of capacity requirements, that is, $CAR_{WS}^{i=1,\ldots,n}$ (Definition 2), and
- returns one output, which is a revised business logic bl_{CAP} that shows the capacities that are necessary to satisfy this set of capacity requirements.

This conversion creates the missing link between the business logic and the capacities through the link that exists between capacities and capacity requirements (Definition 2). The conversion function is as follows (Definition 7).

https://doi.org/10.1017/S0269888914000095 Published online by Cambridge University Press

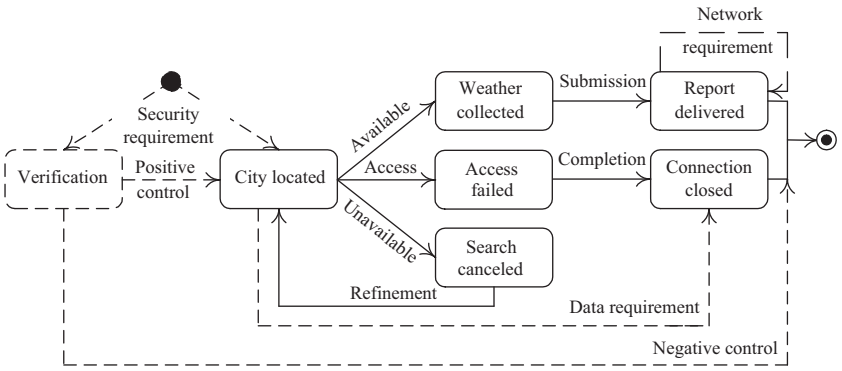


Figure 4 Reviewed business logic of *WeatherWS*

DEFINITION 7 **(Conversion Function).** Let bl and CAR be the business logic of a Web service and the set of all possible capacity requirements on this Web service, respectively. The conversion function is defined as follows: $Conv : bl \times 2^{CAR} \rightarrow bl_{CAP}$. □

The conversion function *Conv* takes a business logic *bl* and a set (possibly empty) of capacity requirements (2^{CAR} is the power set of CAR) and produces a revised business logic bl_{CAP}. This process is done manually, that is, designer driven. An illustration of a revised business logic is given in Figure 4.

Because of the conversion exercise, the initial **goal**, which identifies the functionality of a Web service, is revised with respect to the new mechanisms (in terms of capacities) that empower this Web service. These mechanisms are used for satisfying the capacity requirements. This empowerment has to be captured and reflected on the initial state chart of the business logic of a Web service and can be implemented through three types of actions: *states to repeat*, *states to add*, and *states to skip* (the outcomes of these actions are represented in Figure 4) with dashed lines.

States to repeat. This occurs when the execution of the actions in a certain state did not permit meeting a certain capacity requirement that is put on the Web service. This is detected by checking the post-conditions ($C\bullet$) of this state, which results in repeating the execution of these actions.

For example, `report-delivered` state in *WeatherWS* will be repeated if the minimum acceptable network-bandwidth requirement drops below a certain threshold (`report-delivered's` $C\bullet$: network bandwidth did not improve). The objective is to guarantee the delivery of weather-forecast report. `Report-delivered` state is now bound to a capacity requirement of type network.

States to add. This occurs when the preparation work that is carried out to execute the actions in a certain state did not permit meeting the capacity requirements that are put on the Web service. This is detected by checking the pre-conditions ($\bullet C$) of this state, which results in adding new states to the state chart of this Web service. The new states will be executed several times (obviously bound to a maximum number) until their post-conditions permit satisfying the pre-conditions of the state they precede.

For example, `city-located` state will be preceded by `verification` state if the minimum security-level requirement is not guaranteed. The goal is to restrict the use of *WeatherWS* in an unsecure environment (`city-located's` $\bullet C$: unsecure environment). Contrarily, `verification` state is simply ignored. `City-located` state is now bound to a capacity requirement of type security.

States to skip. This occurs when the number of times that the execution of the actions in a certain state reaches the maximum following the continuous unsatisfaction of a certain capacity requirement that is put on the Web service. This is detected by checking the post-conditions ($C\bullet$) of this state, which results in skipping some of the next states and make the Web service take on appropriate states.

https://doi.org/10.1017/S0269888914000095 Published online by Cambridge University Press

For example, `weather collected`, `access-failed`, and `search canceled` states will be ignored if `city-located` state cannot satisfy the minimum freshness-level requirement of the data it collects out of the database. (`city-located's` $C\bullet$: data freshness is 2 days old). The objective is to guarantee the use of up-to-date data. `City-located` state is now bound to a capacity requirement of type data.

Note. The fact of repeating, adding, or skipping states[4] impacts the set of transitions that are established in the initial state chart of the business logic of the Web service (Figure 3). As a result, new transitions are considered if needed.

Figure 4 shows now the reviewed state chart of *WeatherWS* after making some changes in its initial state chart (Figure 3). These changes show some requirement types that are anchored to some states in this Web service.

DEFINITION 8 **(Web Service Reviewed Goal).** The reviewed goal of a Web service RG_{WS} complies with the definition of a goal as per Section 3.4. □

While the functionality of a Web service remains the same, the business logic is reviewed $RG_{WS} = (fct, rbl)$ like Figure 4 illustrates.

DEFINITION 9 **(Web Service Reviewed Business-Logic).** The state chart of the reviewed business-logic a Web service $RSC_{WS(bl)}$ complies with the definition of a state chart as per Section 3.4. □

4 Capacity-driven Web services deployment

4.1 Description

This section discusses the deployment of capacity-driven Web services following their engineering. In Figure 5[5], a composition scenario (e.g. cookout party) is shown with two Web services: WS_1(*CateringWS*) with `foodCatering` as a functionality and WS_2(*GuestWS*) with `guest-Contact` as a functionality. Both WS_1 and WS_2 need to satisfy some composition requirements (Section 3.2.1) before they participate in this scenario. At run-time, these Web services execute their respective functionalities by selecting and activating the appropriate capacity with respect to the current environment requirements (Definition 2). The deployment of capacity-driven Web services goes through four steps:

- **Step 1** defines the functionalities of Web services, the capacities of each Web service that implement these functionalities, and the requirements under which these Web services participate in compositions and these capacities are activated.
- **Step 2** searches for the relevant Web services based on their respective functionalities. This step is user need-driven and UDDI-based (or any other type of registry).
- **Step 3** checks with the Web services identified in **Step 2** if they would be interested in participating in a composition scenario. Prior to replying either positively or negatively, these Web services verify if they could meet the current composition requirements. A stated earlier an example of composition requirement is the maximum number of compositions to take part in at a time.
- **Step 4** prepares the Web services of **Step 3** for execution. This means identifying the execution requirements (or environment assessment) to satisfy so that the appropriate capacities in these Web services are activated. As stated earlier an example of capacity requirement is data

[4] State skipping could be used to implement state deletion if needed.

[5] In Figure 5, *Restrict/[Comp]* means that requirements are here to restrict the composition scenarios in which a Web service could take part.

https://doi.org/10.1017/S0269888914000095 Published online by Cambridge University Press

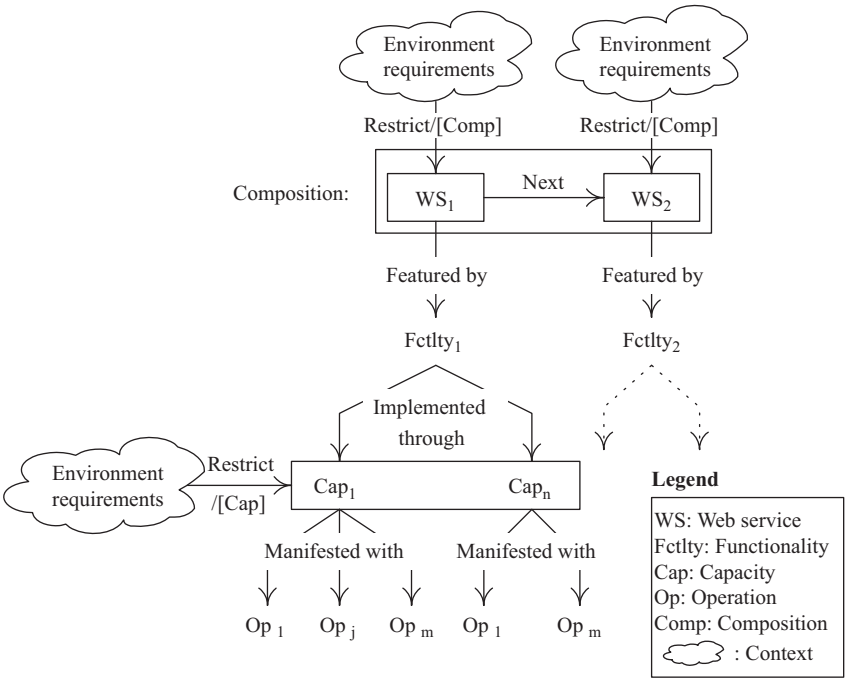


Figure 5 Engineering capacity-driven Web services deployment

freshness. It is noted that any activated capacity in a Web service can be disabled if this capacity can no longer satisfy the current requirements due to changes in the environment. A sudden decrease in network throughput results in first, disabling the current activated capacity in a Web service and second, screening other capacities in this Web service for possible activation as per **Step 4**.

When these four steps are completed, the composition scenario of Figure 5 could be implemented as follows where '→' stands for 'next': $WS_1(Cap_1{:}Op_1,Op_j,Op_m) \rightarrow WS_2(Cap_2{:}Op_1)$. Another configuration like $WS_1(Cap_2{:}Op_1,Op_m) \rightarrow WS_2(Cap_2{:}Op_1)$ can be adopted but this is not the case due to the current requirements. Current practices in the field of Web services neither specialize the operations to execute nor confine these operations into dedicated modules. These operations are, however, usually woven into and scattered across the business logic of a Web service. As a result, this makes any change in both business logic and operations extensive, expensive, and error-prone (Ortiz *et al.*, 2006).

4.2 Deployment

Figure 6 illustrates the managers in the prototype that implements capacity-driven Web services with focus on Steps 2 and 3 (Section 4.1). The composition manager helps designers define composition scenarios. The discovery manager identifies the Web services that satisfy users' needs by searching the *C*-WSDL service registry. Finally, the requirement manager assesses the environment in terms of data, resource, and network prior it sends the details collected back to either the composition or the discovery manager.

When the composition manager receives a user's request (Figure 6, action 1), it asks the discovery manager to look for the component Web services that could take part in the composition scenario of this request (action 2). These component Web services are established after the discovery manager screens the *C*-WSDL service registry (actions 3 and 4). Afterwards, the discovery manager asks the requirement manager to assess the current environment so that

https://doi.org/10.1017/S0269888914000095 Published online by Cambridge University Press

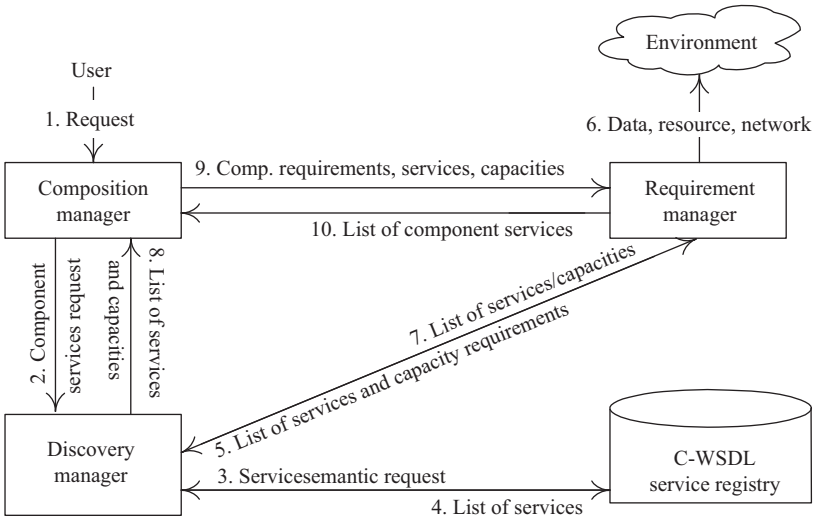


Figure 6 Prototype architecture with focus on discovery and composition steps

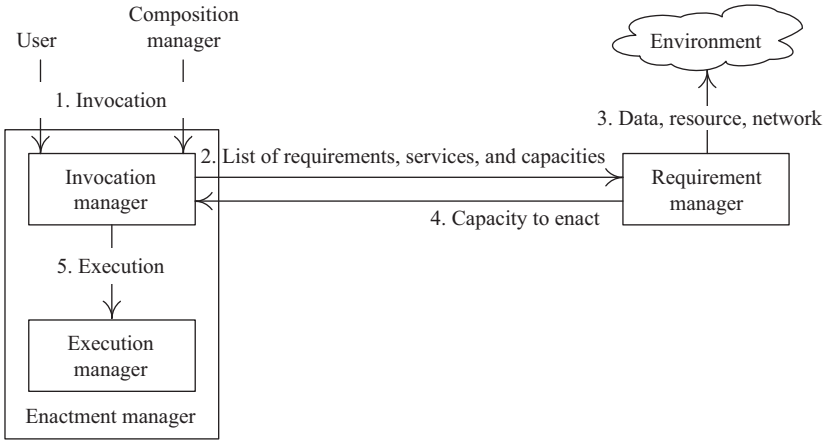


Figure 7 Prototype architecture with focus on the enactment step

data-, resource-, and network-related requirements are identified (actions 5, 6, 7). This would identify the capacities to use per component Web service and per type of requirement. When the composition manager receives the list of component Web services and capacities (action 8) from the discovery manager, it selects the component Web services that will participate in the composition using the composition requirements that the requirement manager sends (actions 9, 10).

Figure 7 details the aforementioned prototype architecture architecture with focus on Step 4 (Section 4.1). Given a selected Web service, the invocation manager identifies the capacity to trigger. The requirement manager assesses the environment in terms of data, resource, and network as described above. The execution manager executes the selected capacity. This execution manager is implemented using Aspect-Oriented Programming (AOP).

5 Conclusion

In this paper, we presented our goal-based approach for the engineering of capacity-driven Web services. Capacities empower Web service with additional 'skills', which make them select the appropriate actions to carry out in response to specific environment requirements. Data freshness,

https://doi.org/10.1017/S0269888914000095 Published online by Cambridge University Press

network bandwidth, and resource reliability are examples of requirements. Our approach sets the goals that initiate the engineering of capacity-driven Web services in terms of requirements, capacities, and business logic. The first goal identifies the types of requirements that affect the way the capacity-driven Web service accomplishes its functionality. The second goal details the capacities that empower and make the capacity-driven Web service satisfy these requirements. Finally, the third goal reviews the business logic that underpins the functionality of the capacity-driven Web service following this capacity empowerment.

The adoption of goals to engineer capacity-driven Web services turns out quite promising. It permits for instance to answer questions related to the types of requirements that can be posed on Web services, the actions per capacity that need to be developed, and the changes that the business logic of a Web service can be subject to. In term of future work, several research opportunities are identified. The first opportunity concerns splitting goals into soft and hard types. The former type would permit to 'twist' goals if Web services are not in a position of satisfying some requirements, which prevent them from reaching these goals. The latter type are fixed and could be used to reinforce the priorities of businesses. The second research opportunity is to examine the feasibility of allowing requirements to evolve over time, which can affect the design and development of capacity-driven Web services.

References

Agre, G., Marinova, Z., Pariente, T. & Micsik, A. 2006. Towards Semantic Web Service Engineering. In *Proceedings of the First International Workshop on Service Matching and Resource Retrieval in the Semantic Web: Issues and Perspectives (SMR'2006)*, Seoul, Korea.

Aoyama, M., Weerawarana, S., Maruyama, H., Szyperski, C., Sullivan, K. & Lea, D. 2002. Web Services Engineering: Promises and Challenges. In *Proceedings of the 2002 International Conference on Software Engineering (ICSE'2002)*, Orlando, Florida, USA.

Arroyo, S., Bussler, C., Kopecký, J. & Lara, R. Auguest 2004. Web Service Capabilities and Constraints in WSMO. Technical report, Digital Enterprise Research Institute (DERI), Galway, Ireland and Innsbruck, Austria.

Birman, K. P. 2004. Like it or not, web services are distributed objects. *Communications of the ACM* **47**(12), 60–62.

Chris Gibson, J. 2004. Developing a requirements specification for a Web service application. In *Proceedings of The 12th IEEE International Requirements Engineering Conference (RE'2004)*, Kyoto, Japan.

Chung, L., Nixon, B. A., Yu, E. & Mylopoulos, J. 1999. *Non-Functional Requirements in Software Engineering*, *International Series in Software Engineering* **5**. Springer.

Damas, C., Lambeau, B. & van Lamsweerde, A. 2006. Scenarios, Goals, and State Machines: a Win-Win Partnership for Model Synthesis. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'2006)*, Portland, Oregon, USA.

Donzelli, P. 2004. A Goal-driven and Agent-based Requirements Engineering Framework. *Requirement Engineering* **9**(1).

Elghazi, H. 2007. A goal-driven method for automated systems requirements engineering. In *Proceedings of the First International Conference on Research Challenges in Information Science (RCIS'2007)*, Ouarzazate, Morocco.

Heiko, L., Toshiyuki, N., Philipp, W., Oliver, W. & Wolfgang, Z. 2006. Reliable Orchestration of Resources using WS-Agreement. Technical Report TR-0050, Institute on Grid Systems, Tools, and Environments, CoreGRID – Network of Excellence.

Jha, A. 2006. Problem frames approach to Web services requirements. In *Proceedings of the 2nd International Workshop on Advances and Applications of Problem Frames (IWAAPF'2006)*, Shanghai, China.

Kazhamiakin, R., Pistore, M. & Roveri, M. 2004. A framework for integrating business processes and business requirements. In *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC'2004)*, Monterey, California, USA.

Kirda, E., Kerer, C., Kruegel, C. & Kurmanowytsch, R. 2003. Web service engineerning with DIWE. In *Proceedings of the 29th EUROMICRO Conference 2003, New Waves in System Architecture (EUROMICRO'2003)*, Belek-Antalya, Turkey.

Maamar, Z., Benslimane, D. & Narendra, N. C. 2006. What can context do for Web services? *Communications of the ACM* **49**(12), 98–103.

Maamar, Z., Tata, S., Belaïd, D. & Boukadi, K. 2009. Towards an approach to defining capacity-driven Web services. In *Proceedings of the 23rd International Conference on Advanced Information Networking and Applications (AINA2009)*, Bradford, UK.

https://doi.org/10.1017/S0269888914000095 Published online by Cambridge University Press

Malay Chatterjee, A., Pal Chaudhari, A., Saurav Das, A., Dias, T. & Erradi, A. 2005. Differential QoS support in Web services management. *SOA World Magazine* **5**(8), 781–788.

Oldham, N., Verma, K., Sheth, A. & Hakimpour, F. 2006. Semantic WS agreement partner selection. In *Proceedings of the 15th International World Wide Web Conference (WWW'2006)*, Edinburgh, Scotland.

Ortiz, G., Hernndez, J. & Clemente, P. J. 2006. Web services orchestration and interaction patterns: an aspect-oriented approach. In *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC'2004)*, New-York, USA.

Robinson, W. N. 2003. Monitoring Web service requirements. In *Proceedings of the 11th IEEE International Requirements Engineering Conference (RE'2003*), Monterey, California, USA.

Tsai, W. T., Jin, Z., Wang, P. & Wu, B. 2007. Requirement engineering in service-oriented system engineering. In *Proceedings of the 2007 IEEE International Conference on e-Business Engineering (ICEBE'2007)*, Hong Kong, China.

van Eck, P. & Wieringa, R. 2004. Web services as product experience augmenters and the implications for requirements engineering: a position paper. In *Proceedings of the International Workshop on Service-oriented Requirements Engineerings (SoRE'2004)*, Kyoto, Japan.

van Lamsweerde, A. 2001. Goal-oriented requirements engineering: a guided tour. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE'2001)*, Toronto, Canada.

Vukovic, M. & Robinson, P. 2004. Adaptive, planning-based, Web service composition for context awareness. In *Proceedings of the 2nd European Conference on Web Services (ECOWS'2004)*, Erfurt, Germany.

Zave, P. & Jackson, M. 1997. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology* **6**(1), 1–30.

https://doi.org/10.1017/S0269888914000095 Published online by Cambridge University Press