ENGINEERING PROCESS FOR CAPACITY-DRIVEN WEB SERVICES

Imen Benzarti, Samir Tata
Institut TELECOM, TELECOM SudParis, CNRS UMR, Samovar, Evry, France

Zakaria Maamar
Zayed University, Dubai, U.A.E.

Nejib Ben Hadj-Alouane, Moez Yeddes
National School of Computer Sciences, Tunis, Tunisia

Keywords: Web service, Capacity-driven Web service, Engineering process.

Abstract: This paper presents a novel approach for the engineering of capacity-driven Web services. By capacity, we mean how a Web service is empowered with several sets of operations from which it selectively triggers a set of operations with respect to some run-time environmental requirements. Because of the specificities of capacity-driven Web services compared to regular (i.e., mono-capacity) Web services, their engineering in terms of design, development, and deployment needs to be conducted in a complete specific way. Our approach define an engineering process composed of five steps: (1) to frame the requirements that could be put on these Web services, (2) to define capacities and how these capacities are triggered, and last but not least link these capacities to requirements, (3) to identify the processes in term of business logic that these Web services could implement, (4) to generate the source code, and (5) to generate the $\mathcal{C}$apacity-driven Web Services Description Language ($\mathcal{C}$-WSDL).

1 INTRODUCTION

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 perspectives[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 entitled to so that, it either accepts or rejects processing clients' requests. A client could be a user or another Web service.

In (Maamar et al., 2009b), we discuss the notion of *capacity* and how it can smoothly be woven into Web services. By capacity, we mean how a Web service is empowered with several sets of operations from which it selectively triggers a set of operations with respect to some run-time environmental requirements. The description of these sets of operations is included in the $\mathcal{C}$apacity-driven Web Services Description Language ($\mathcal{C}$-WSDL) document of the Web service. In this paper, we continue our discussions on capacity-driven Web services with focus this time on how one could engineer such Web services upon which enterprise applications could be built. The rationale of examining capacities is backed by the claim 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 qualify

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

the Web services in (Malay Chatterjee et al., 2005) as *mono-capacity*.

Capacity is an intrinsic element of the approach we developed in (Maamar et al., 2009b) in response to the specificities that characterize and the challenges that underpin each of the aforementioned perspectives. In this approach, we decided to let Web services first, assess themselves before they accept to participate in any composition scenario, and then identify 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/disappearance without prior notice, and computing resources mobility and sometimes failure.

The deployment of capacity-driven Web services sheds the light on 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 needed assistance to service engineers. A part of this assistance is usually known as requirement engineering in the software field (Sommerville, 2001). Our contributions in this paper are manifold: define an engineering process for 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 Web services can satisfy them through capacities; and, propose techniques for the generation of the code and the description of capacity-driven Web services.

This paper is organized as follows. Section 2 presents motivating example, provides an overview of the related work and recalls the definition of capacity-driven Web services. Section 3 define our engineering process for capacity-driven Web services. Finally, conclusions are drawn in Section 4.

2 BACKGROUND

This section consists in four parts. The first part presents a running example that we will use in this paper for illustration purpose. The second present a review of the related work. The third part of this section recalls the definitions of Capacity-driven Web services. The last part presents environment assessment in service composition and invocation.

2.1 Running Example

Our running example concerns a real-state office that runs and sells different types of properties such as villas and flats. The office is equipped with a set of PCs, while the staff in charge of conducting the visits are equipped with various heterogeneous handheld devices. The staff are usually in contact with customers as per the following description. The customers contact the office to request an estimate, purchase, or rent a property. The office can also contact customers as per their initial requests.

“Get the map of properties in the vicinity” is among the services that the real-state staff use in their day-to-day business. This service returns all the properties that the office manages and are within a walking distance from a staff. This distance and other criteria like price range and number of bedrooms are set by the staff. “Get the map of properties in the vicinity” is treated as a composite Web service that relies on internal and external Web services as per the following description:

1. “LocateAgent” returns the current position of a staff in terms of latitude and longitude.
2. “LocatetProperties” retrieves the list of properties (identifier, location, etc.) that are in the vicinity of the staff (at a maximum distance from her) and whose characteristics match her criteria.
3. “GenerateMap” produces a navigation map that shows some properties in response to the staff’s request.
4. “Display” displays the map on the device of the staff.

It is worth mentioning that “LocateAgent” and “GenerateMap” services are context aware. Their performance depends on the characteristics of the staff’s device: network interfaces type (GSM, WiFi, GPS, etc.), quality of the signal strength of the connexion, screen size, etc.

2.2 Related Work

Our literature review identified a good number of research initiatives that looked into the engineering Web services (Chris Gibson, 2004; Jha, 2006; Kirda et al., 2003; Robinson, 2003; Tsai et al., 2007; van Eck and Wieringa, 2004). Unfortunately none of these initiatives is adapted to capacity-driven Web serivces. Furthermore, our literature review included some other initiatives on requirement engineering like (Damas et al., 2006; Donzelli, 2004; 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; it simplifies application development and reduces development risk and cost. The requirement engineering process of Chris Gibson is commonly adopted by the IT community through the steps of 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 a business so that, the objectives, needs, and context of this business 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 build such systems (Zave and Jackson, 1997). To capture an organization's objective, needs, and context, Jha suggests two steps: understand the organization's business strategy and overall objective, and use progression of problems to describe this organization'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, they 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 (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 because 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.

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 type of Web services supports other services that are not themselves Web services. van Eck and Wieringa use the wording of product experience augmenters to qualify such Web services. The authors 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 & sales department or the departments that offer the primary product, and they need to properly fit into the business processes.

2.3 Capacity-driven Web Services

Capacity is an aggregation of a set of actions that implement the functionality of a Web service (Maamar et al., 2009b).. Functionality (e.g., currencyConversion) is usually used to differentiate a Web service from other peers, though it is common that independent bodies develop Web services with similar functionalities but different non-functional properties (Bui and Gacher, 2005; Maamar et al., 2009a). Concretely speaking, the actions in a capacity correspond to operations in a WSDL document and vary according to the business-application domain. As per our proposed running example, the actions in LocateProperties Web service could be:

- Retrieve a list of properties' identifiers in the vicinity of the staff using the coordinates (latitude, longitude) of the staff and the maximum distance.
- Select from the list of identifiers the properties that satisfy the staff's requirements.
- Locate (latitude, longitude) the identified properties.

In Fig. 1 we illustrate a simple specification of the real-state office service using a finite state machine (Harel and Naamad, 1996). The component Web services that agreed to participate in this composition are "LocateAgent", "LocateProperties", "GenerateMap" and "Display". If one of these Web services rejects the participation, the discovery step is reactivated again.

Several capacities along with their respective actions satisfy the unique functionality of a Web service in different ways. Which capacity to select and make active out of the available capacities in a Web service requires assessing the environment so that appropriate details are collected and prepared for this selection. Depending on the characteristics of the staff's device, "LocateAgent" can be carried out with one of the following capacities:

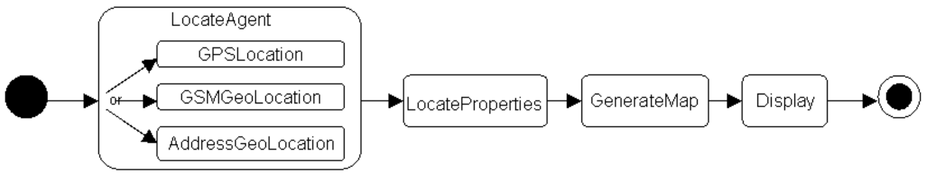


Figure 1: Specification of the real-state office example.

- “GPSGeoLocation” capacity that is an internal function of the device if the device is equipped with a Global Positioning System (GPS).
- “GSMGeoLocation” capacity that provides the position of the staff’s device if the device is equipped with a Global System for Mobile (GSM).
- “AddressGeoLocation” capacity that provides the position (latitude, longitude) of the staff from the standardized address.

2.4 Environment Assessment

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

3 ENGINEERING CAPACITY-DRIVEN WEB SERVICES

In this section, we answer the question: How to develop a capacity-driven Web service. The development of a monocapacity Web service can be done in two different ways: The first is to generate the skeleton of the code source service from the WSDL specification before completing it with the business code, the second one is to generate WSDL specification from a web service code source before deploying the Web service.

Because of the specificities of capacity-driven Web services compared to regular (i.e., mono-capacity) Web services, their engineering in terms of design, development, and deployment needs to be conducted in a complete specific way. Our approach define an engineering process composed of five steps: (1) to frame the requirements that could be put on these Web services, (2) to define capacities and how these capacities are triggered, and last but not least link these capacities to requirements (3) to identify the processes in term of business logic that these Web services could implement (4) to generate the source code and (5) to generate the $\mathcal{C}$apacity-driven Web Services Description Language ($\mathcal{C}$-WSDL).

As prsented in Fig. 2, the proposed engineering process for capacity-driven Web services is composed of five steps:

1. Step 1: requirement engineering,
2. Step 2: capacity engineering,
3. Step 3: business logic engineering,
4. Step 4: source Code generation,
5. step 5: CWSDL description gnration

In the following, these five steps are detailed. We use the example presented in Section 2.1 to illustrates these steps.

3.1 Requirement Engineering

In this first step of the engineering approach, the goal is to contribute towards (i) reflecting the dynamic nature of the environment, (ii) understanding the types of requirements that this environment poses on Web services, and last but not least (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” , we identify two goals and associate them with “composition requirements” and “capacity requirements”, respectively. The former goal is meant to secure the participation of a Web service in a new composition scenario, and the latter goal is meant to secure the activation of at least one capacity in a Web service at run-time.

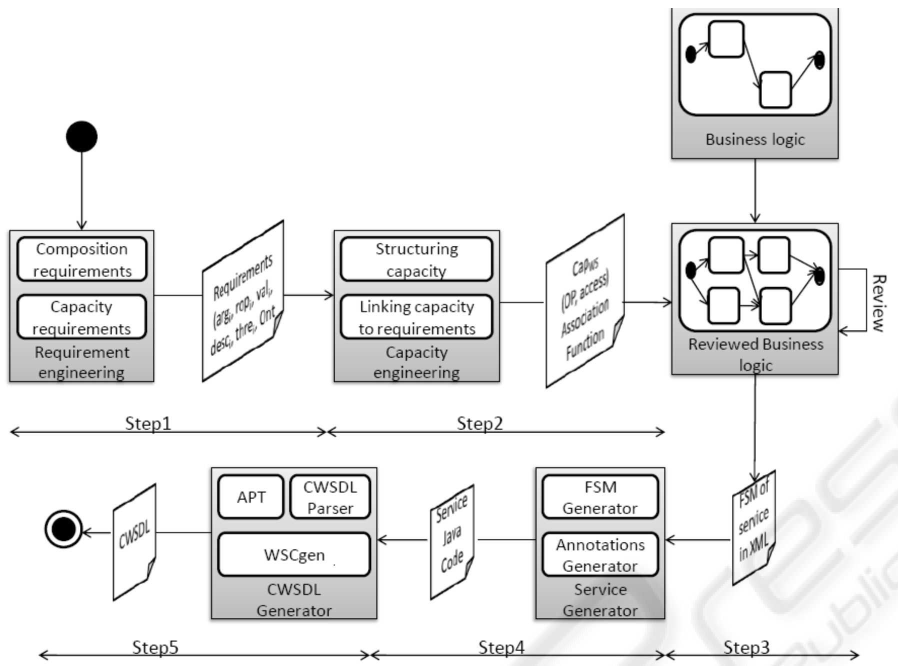


Figure 2: Engineering process for capacity-driven Web services.

3.1.1 Composition Requirements

In (Maamar et al., 2006), we introduced the Web services instantiation principle that illustrates 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 organized along three dimensions: past compositions, current compositions, and forthcoming compositions. Requirements apply to forthcoming compositions and thus, could limit the participation of a Web service in these compositions.

Definition 1 (**Composition Requirement**). The composition requirement on a Web service is a tuple $\mathcal{COR}_{\mathcal{WS}} = (arg_i, rop_i, val_i, desc_i, Ont)$ where:

- arg_i is the name of a type of composition requirement.
- rop_i is a relational operator ($=, <, \leq, >, \geq$, or $\neq$).
- 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 $\mathcal{COR}$.

Examples: *The following elements could populate $\mathcal{COR}$:*

- *arg_1: number of participations, rop_1: $\leq$, val_1: 10, $desc_1$: the maximum number of participation for a Web service in compositions at a time should not exceed 10, Ont: $ONT_{comp-req}$.*
- *arg_2: date of maintenance, rop_2: $=$, 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: $ONT_{comp-req}$.*

3.1.2 Capacity Requirements

A Web service implements the functionality it offers through capacities. Which capacity to activate at runtime depends on which requirements are put on this Web service. We classify requirements on capacities into different types with focus in this paper on the following three types (additional ones could be added, as need be):

1. Data requirement is about the quality of data that a Web service receives, manipulates, and probably sends out. Such a requirement could be about freshness (when were data obtained), source (who is the sender of data), and validity (when do 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. Such a requirement could be about bandwidth, throughput, and reliability.

3. Resource requirement is about the computing facilities upon which the performance of a Web service is scheduled. Such a requirement could be about availability and reliability.

Definition 2 (**Capacity Requirement**). The capacity requirement on a Web service is a tuple $\mathcal{CAR}_{\mathcal{WS}} = (arg_i, rop_i, val_i, desc_i, thre_i, Ont)$ where:

- arg_i is the name of a capacity requirement.
- rop_i is a relational operator ($=, <, \leq, >, \geq$, or $\neq$).
- val_i is a value (numerical, string, etc.) assigned to arg_i.
- $desc_i$ 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 affect val_i.
- Ont refers to the ontology defining arg_i.

The set of all possible capacity requirements is denoted by $\mathcal{CAR}$.

Examples: *The following elements could populate* $\mathcal{CAR}$*:*

- *arg_1: data freshness, rop_1: $=$, val_1: current, $desc_1$: all data to be used by a Web service need to be of today, $thre_1$: null, Ont: $ONT_{cap-req}$.*
- *arg_2: resource availability, rop_1: $\leq$, val_2: 80%, $desc_2$: resources upon which a Web service runs, need to have an 80% availability, $thre_2$: 5% decrease in resource availability is still acceptable to the Web service and the current activated capacity can continue to be used, Ont: $ONT_{cap-req}$.*

3.2 Capacity Engineering

In this second step of the engineering approach, the goal is to contribute towards (i) understanding the different ways of structuring 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 2.3, we identify goals with respect to the multiple 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 can satisfy.

Definition 3 (**Web Service Capacity**). The capacity in a Web service is a couple $\mathcal{CAP}_{\mathcal{WS}} = (\mathcal{OP}, access)$ where:

- $\mathcal{OP}$ is a set of operations. An operation $Op_i \in \mathcal{OP}$ consists of a name N, a set of input arguments $Input$ along with their appropriate types, and a set of output arguments $Ouput$ along with their appropriate 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 $\mathcal{CAP}$. The rationale of public and private access modifiers is given later.

Example: *Let us consider $\mathcal{CAP}^1_{PlaceBookingWS}$ 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 $\mathcal{WS}$ for triggering is subject to satisfying some capacity requirements, i.e., $\mathcal{CAR}^{i=1,\ldots,n}_{\mathcal{WS}}$. As a result, we define a function that associates a capacity with capacity requirements as per Definition 4.

Definition 4 (**Association Function**). Let $\mathcal{CAP}$ and $\mathcal{CAR}$ 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: $Assoc : \mathcal{CAP}_{\mathcal{WS}} \rightarrow 2^{\mathcal{CAR}}$.

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

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

Based on Definition 3 and Definition 4, the following comments are made:

- A capacity in $\mathcal{WS}$, which does not have any requirement to satisfy is referred to as *default* capacity, i.e., $Assoc(\mathcal{CAP}^i_{\mathcal{WS}}) = \emptyset$. This capacity is automatically selected when none of the existing capacities in $\mathcal{WS}$ satisfies the current capacity requirements. By having a default capacity, $\mathcal{WS}$ can always guarantee a user's request satisfaction.
- Independent capacities in $\mathcal{WS}$ could have common operations, i.e., $\cap^n_{i=1} \mathcal{CAP}^i_{\mathcal{WS}} \neq \emptyset$.

- Common operations in independent capacities could be overloaded if needed. For example, Op_1 could take one input argument in $\mathcal{CAP}^1_{\mathcal{WS}}$, e.g., in(username:String), and two input arguments in $\mathcal{CAP}^n_{\mathcal{WS}}$, e.g., in(username:String, password:String).

In Definition 3, capacity has either public or private access-modifier like in object-oriented programming languages (Fig. 3)[2]. On the one hand, public capacities are offered to the external environment particularly to 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 needed ((0,n) cardinality in Fig. 3), but this is not the case with public capacities ((0,0) cardinality in Fig. 3) since they all implement the unique functionality of a Web service.

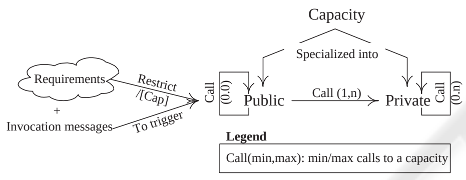


Figure 3: Capacity categorization.

3.3 Business Logic Engineering

In this third step of the engineering approach, the goal is to contribute 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 changes in processes to requirement satisfaction.

As per the definition of "Web service functionality" in Section 2.3, 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 different 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 $\mathcal{G}_{\mathcal{WS}} = (fct, bl)$ where:

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

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 $\mathcal{S}$tate $\mathcal{C}$hart $\mathcal{SC}$ of the business logic of a Web service is a tuple $\mathcal{SC}_{\mathcal{WS}(bl)} = (\mathcal{S}, \mathcal{S}^{\mathcal{F}}, \mathcal{L}, \mathcal{T}, s^0)$ where:

- $\mathcal{S}$ is a finite set of state names.
- $s^0 \in \mathcal{S}$ is the initial state in $\mathcal{SC}_{\mathcal{WS}}$.
- $\mathcal{S}^{\mathcal{F}} \subset \mathcal{S}$ is a finite set of final states.
- $\mathcal{L}$ 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 Fig. 4, we just name labels without giving full details on conditions and actions.

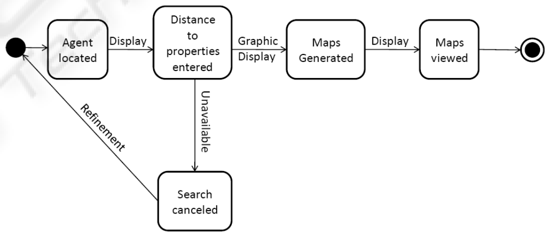


Figure 4: Business logic of the real-state office example.

In our 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}, i.e., $s_i(C\bullet) \Rightarrow s_{i+1}(\bullet C)$.

- $\mathcal{T} \subseteq (\mathcal{S} \times \mathcal{L} \times \mathcal{S})$ is a finite set of transitions. Each transition $t \in \mathcal{T}, t = (s^{src}, l, s^{tgt})$ consists of a source state $s^{src} \in \mathcal{S}$, a target state $s^{tgt} \in \mathcal{S}$, and a transition label $l \in \mathcal{L}$.

Example: *Fig. 4 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 various transitions such*

[2]In Fig. 3, *Restrict/[Cap]* means that requirements are here to restrict the capacity that a Web service could select. And, *Call(min,max)* means the minimum and maximum times that a capacity could 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)*.

as (city-located, <u>unavailable</u>, search-canceled) where city-located and search-canceled are the source and target states, respectively, and <u>unavailable</u> is the label of the transition.

The business logic in Fig. 4 is an example of how a "regular" (i.e., mono-capacity) Web service for instance *WeatherWS* would function independently of the requirements of type capacity (could be of related to security, network, trust, etc.) that could be posed on this Web service, and thus could 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 need to be handled nor how this functionality might be 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, i.e., $\mathcal{CAR}_{\mathcal{WS}}^{i=1,..,n}$ (Definition 2), and
- returns one output, which is a revised business logic $bl_{\mathcal{CAP}}$ that shows the capacities that are needed to satisfy this set of capacity requirements.

This way of doing creates the missing link between business logic and capacities through the link that exists between capacities and capacity requirements (Definition 4). The conversion function is as follows (Definition 7).

Definition 7 (**Conversion Function**). Let *bl* and $\mathcal{CAR}$ 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: $Conv : bl \times 2^{\mathcal{CAR}} \rightarrow bl_{\mathcal{CAP}}$.

The conversion function *Conv* takes a business logic *bl* and a set (possibly empty) of capacity requirements ($2^{\mathcal{CAR}}$ is the power set of $\mathcal{CAR}$) and produces a revised business logic $bl_{\mathcal{CAP}}$. This process is done manually, i.e., designer driven. An illustration of a revised business logic is given in Fig. 5.

Because of the conversion exercise, the initial goal, which identifies the functionality of a Web service, is reviewed with respect to the additional 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 could be implemented through three types of actions: *states to repeat*, *states to add*, and *states to skip* (the outcomes of these actions are represented with dashed lines in Fig. 5).

- **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.

  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[3] impacts the set of transitions that are established in the initial state chart of the business logic of the Web service (Fig. 4). As a result, new transitions are considered if needed.

Fig. 5 shows now the reviewed state chart of *WeatherWS* after making some changes in its initial state chart (Fig. 4). 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 $\mathcal{RG}_{\mathcal{WS}}$ complies with the definition of a goal as per Definition 5.While the functionality of a Web service remains the same, the business logic is reviewed $\mathcal{RG}_{\mathcal{WS}} = (fct, rbl)$ like Fig. 5 illustrates.

Definition 9 (**Web Service Reviewed Business-Logic**). The state chart of the reviewed business-logic a Web service $\mathcal{RSC}_{\mathcal{WS}(bl)}$ complies with the definition of a state chart as per Definition 6.

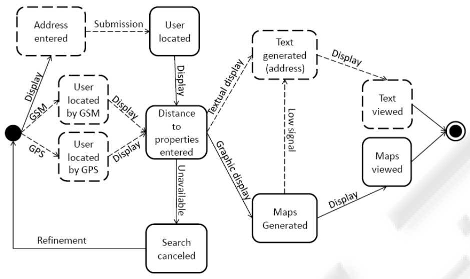


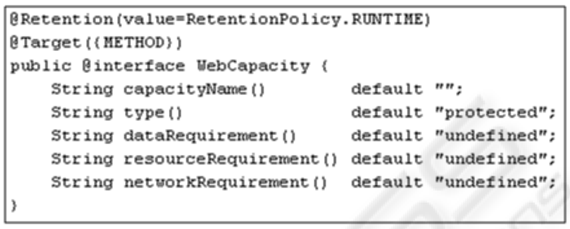
Figure 5: Reviewed business logic of GeoLocProp.

3.4 Source Code Generation

Our goal is to generate java code annotated with different capacities from the reviewed statechart. This led us to study the generator of Java code from statecharts diagrams (final state machine: FSM)

3.4.1 The Annotation of Java Code with Capacity

Many APIs require a fair amount of boilerplate code. For example, in order to write a JAX-RPC web service, we must provide a paired interface and implementation. This boilerplate could be generated automatically by a tool if the program were ”decorated” with annotations indicating which methods were remotely accessible. Annotations play a critical role in JAX-WS 2.0 (the successor of JAX-RPC API). First, annotations are used in mapping Java to WSDL and schema. Second, annotations are used a runtime to control how the JAX-WS runtime processes and responds to web service invocations. In our case we are interested to the mapping between java and WSDL. We propose new annotation that will process the transformation from java to WSDL annotated with capacity (CWSDL).



```
@Retention(value=RetentionPolicy.RUNTIME)
@Target({METHOD})
public @interface WebCapacity {
    String capacityName()        default "";
    String type()                default "protected";
    String dataRequirement()     default "undefined";
    String resourceRequirement() default "undefined";
    String networkRequirement()  default "undefined";
}
```

Figure 6: Implementation of interface WebCapacity.

Figure 6 presents the implementation of the interface WebCapacity that takes as parameters the name capacity, visibility, requirements of type data, resources and network.

3.4.2 Java Code Generators from FSM

According to our study of the generator FSM, we distinguish two types of generator. (i) Generators that generate the machine into a single class that presents the business logic like FSMGneretor (Gurp and Bosch, 1999) (ii) generator that generates a java class for each state(Mura and Sami, 2008). On the other hand, the specification of departure may be (i) a java class (Sakharov, 2000) or (ii) in XML (Gurp and Bosch, 1999).

Our goal is to generate java code from a state machine that describes the behavior of a web service and annotated with the capacity for this service. The generated code must represent the business logic of web service by a single Java class. The generation must therefore be done in one class. On the other hand, the specification of the state machine must be simple and consistent with the definition of a state machine . The specification is most consistent with the XML specification (Figure 7) ; the other code generated by this tool is one class that describes the business logic of the state machine.

3.4.3 Code Generator

Generating Java code from state machine of the web service described in XML is based on the generator FSMGenerator. The generated code also contains the annotation @WebService and @WebCapacity.

Capacity requirements are extracted from the transitions added in the state diagram transition revised.

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

```
<machine id="GeoLocProp" name="GeoLocProp">
    <start id="S0" name="Start"/>
    <states>
        <state id="S0" name="Start" isFinal="false"/>
        ...
        <state id="S10" name="Map_viewed" isFinal="false"/>
    </states>
    <events>
        <event id="E1" name="Display"/>
        ..
        <event id="E9" name="Low_signal"/>
    </events>
    <transitions>
        <transition id="T1"  name="T1"  from="Start" on="Display" to="Adress_entered"/>
        ...
        <transition id="T14" name="T14" from="Text_generated" on="Display" to="Text_viewed"/>
    </transitions>
    <callbacks>

    </callbacks>
    <hooks>

    </hooks>
</machine>
```

Figure 7: Specification of FSM of GeoLocProp in XML.

To associate these transitions with the type of capacity requirements (data, resources, network) a model is used. This led us to look into the existing approach for requirement modeling. Modeling of requirements must be divided into three subtypes (data, resources, network) this has led us to study models of device capability.

As the number of user devices is proliferating and as the capabilities of the devices are augmenting on daily basis, there have been a number of different approaches for describing devices capability. The CC/PP (Kim and Lee, 2006)profile describes the capabilities of the device and, possibly, the preferences of the user. It serves a model and provides core vocabulary for the devices' capabilities description. The extended CC/PP presented in (Mukhtar et al., 2008) model classifies device capability into hardware, software and network categories. (Bandara et al., 2004) introduces ontology for devices description intended at providing a general framework to describe any type of device. The information related to a device is logically divided into five classes depending on the type of information they provide: namely Device Description, Hardware Description, Software Description, Device Status (including location) and Service. These approaches have common characteristics that they are both based on languages designed for semantic matching purposes (OWL and RDF). We rely on these models to create a modeling for capacity requirements (Figure 8) . We associate the data level to software level while adding other concepts, resources level to hardware level and network level is already in the models already presented.

To define the different types of capacity in FSM the generator uses this model to associate each transition to capacity in order to generate the annotation in

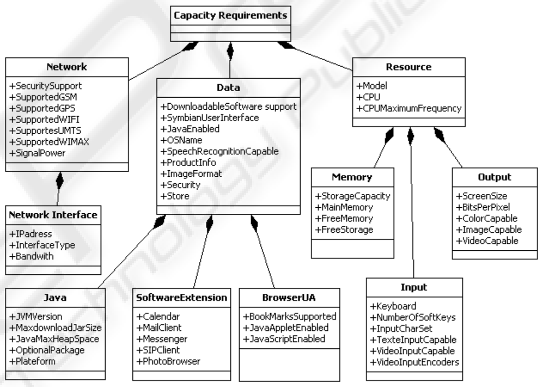


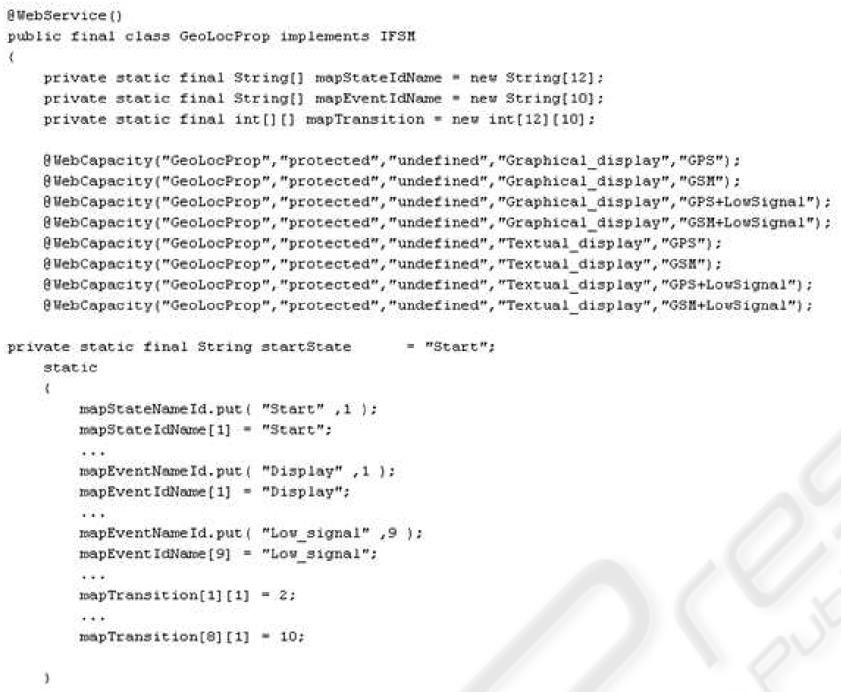
Figure 8: Modelling for capacity requirements.

the Java code. Part of the generated code is presented in Fig Figure 9.

3.5 CWSDL Description Generation

CWSDL generation is done with the same process as generating WSDL. We studied the source code for the JAX-WS API. This API uses the tool wsgen that interprets different annotations in java code and generates the wsdl while relying on the XML schema for WSDL.

To generate the CWSDL by JAX-WS we went through the following steps: (1)development of the class interface WebCapacity (2)replace the XML schema of the WSDL CWSDL (3)modified the tool wsgen that it interprets the new annotation. The Figure 10 present a fragment of the generated cwsdl for the service GeoLocProp.



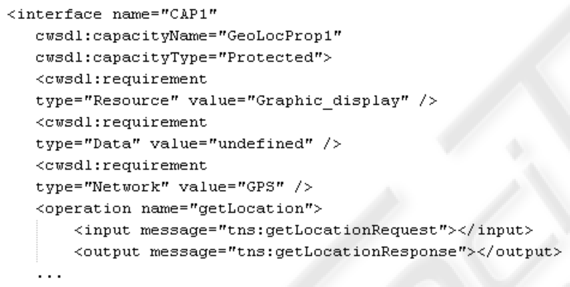
```
@WebService()
public final class GeoLocProp implements IFSM
{
    private static final String[] mapStateIdName = new String[12];
    private static final String[] mapEventIdName = new String[10];
    private static final int[][] mapTransition = new int[12][10];

    @WebCapacity("GeoLocProp","protected","undefined","Graphical_display","GPS");
    @WebCapacity("GeoLocProp","protected","undefined","Graphical_display","GSM");
    @WebCapacity("GeoLocProp","protected","undefined","Graphical_display","GPS+LowSignal");
    @WebCapacity("GeoLocProp","protected","undefined","Graphical_display","GSM+LowSignal");
    @WebCapacity("GeoLocProp","protected","undefined","Textual_display","GPS");
    @WebCapacity("GeoLocProp","protected","undefined","Textual_display","GSM");
    @WebCapacity("GeoLocProp","protected","undefined","Textual_display","GPS+LowSignal");
    @WebCapacity("GeoLocProp","protected","undefined","Textual_display","GSM+LowSignal");

private static final String startState      = "Start";
    static
    {
        mapStateNameId.put( "Start" ,1 );
        mapStateIdName[1] = "Start";
        ...
        mapEventNameId.put( "Display" ,1 );
        mapEventIdName[1] = "Display";
        ...
        mapEventNameId.put( "Low_signal" ,9 );
        mapEventIdName[9] = "Low_signal";
        ...
        mapTransition[1][1] = 2;
        ...
        mapTransition[8][1] = 10;

    }
```

Figure 9: The generated code for GeoLocProp.



```
<interface name="CAP1"
   cwsdl:capacityName="GeoLocProp1"
   cwsdl:capacityType="Protected">
   <cwsdl:requirement
   type="Resource" value="Graphic_display" />
   <cwsdl:requirement
   type="Data" value="undefined" />
   <cwsdl:requirement
   type="Network" value="GPS" />
   <operation name="getLocation">
       <input message="tns:getLocationRequest"></input>
       <output message="tns:getLocationResponse"></output>
   ...
```

Figure 10: A fragment of the generated cwsdk for the service GeoLocProp.

4 CONCLUSIONS

In this paper we presented the concepts, definitions, issues, and solutions that underpin the design, development, and deployment of capacity-driven Web services. We recalled the concept of capacity as a new way of making Web services take appropriate actions in response to some environmental requirements. Then we defined a novel approach for engineering capacity-driven Web services composed of five steps: to frame the requirements, to define capacities, to identify the business logic, (4) to generate the source code, and (5) to generate the $\mathcal{C}$apacity-driven Web Services Description Language ($\mathcal{C}$-WSDL).

In term of future we identified different research issues that need to be addressed such as requirement consistency and capacity types and Capacity-driven web service deployment based on the extension of Web service containers .

REFERENCES

Bandara, A., Payne, T. R., de Roure, D., and Clemo, G. (2004). An ontological framework for semantic description of devices. In *International Semantic Web Conference (ISWC)*.

Bui, T. and Gacher, A. (2005). Web Services for Negotiation and Bargaining in Electronic Markets: Design Requirements and Implementation Framework. In *Proceedings of The 38th Hawaii International Conference on System Sciences (HICSS'2005)*, Big Island, Hawaii, USA.

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.

Damas, C., Lambeau, B., and 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, Springer*, 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.

Gurp, J. V. and Bosch, J. (1999). On the implementation of finite state machines. In *in Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications, IASTED/Acta*, pages 172–178. Press.

Harel, D. and Naamad, A. (1996). The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4).

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.

Kim, H.-M. and Lee, K.-H. (2006). Device-independent web browsing based on cc/pp and annotation. *Interact. Comput.*, 18(2):283–303.

Kirda, E., Kerer, C., Kruegel, C., and 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., and Narendra, N. C. (2006). What Can Context do for Web Services? *Communications of the ACM*, 49(12).

Maamar, Z., Subramanian, S., Bentahar, J., Thiran, P., and Benslimane, D. (2009a). An Approach to Engineer Communities of Web Services - Concepts, Architecture, Operation, and Deployment . *International Journal of E-Business Research, IGI Global*, 5(4).

Maamar, Z., Tata, S., Belaïd, D., and Boukadi, K. (2009b). Towards An Approach to Defining Capacity-Driven Web Services. In *Proceedings of the 23rd International Conference on Advanced Information Networking and Applications (AINA'2009)*, Bradford, UK.

Malay Chatterjee, A., Pal Chaudhari, A., Saurav Das, A., Dias, T., and Erradi, A. (2005). Differential QoS Support in Web Services Management. *SOA World Magazine*, 5(8).

Mukhtar, H., Belaïd, D., and Bernard, G. (2008). A model for resource specification in mobile services. In *SIPE '08: Proceedings of the 3rd international workshop on Services integration in pervasive environments*, pages 37–42, New York, NY, USA. ACM.

Mura, M. and Sami, M. G. (2008). Code generation from statecharts: Simulation of wireless sensor networks. In *DSD '08: Proceedings of the 2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, pages 525–532, Washington, DC, USA. IEEE Computer Society.

Robinson, W. N. (2003). Monitoring Web Service Requirements. In *Proceedings of The 11th IEEE International Requirements Engineering Conference (RE'2003)*, Monterey, California, US.

Sakharov, A. (2000). A hybrid state machine notation for component specification. *SIGPLAN Not.*, 35(4):51–56.

Sommerville, I. (2001). *Software Engineering, 6th Edition*. Addison-Wesly Publishers Limited.

Tsai, W. T., Jin, Z., Wang, P., and 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. and 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.

Zave, P. and Jackson, M. (1997). Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1).