Journal of the Brazilian Computer Society
a SpringerOpen Journal

RESEARCH Open Access

A path to automated service creation via semi-automation levels

Ernani Azevêdo[1*], Carlos Kamienski[2], Ramide Dantas[1], Börje Ohlman[3] and Djamel Sadok[1]

Abstract

The manual creation of new Internet- and IT-based applications is currently a limiting factor in enabling new and innovative services to be quickly available. We advocate that semi-automated service creation techniques are feasible, whereas fully automated ones are not a reality yet. Consistently increasing the level of automation may lead to a better comprehension of the problem that will pave the way for the introduction of higher levels of automation in the future. We have developed two versions of a service creation tool, with different levels of automation, which have so far confirmed our expectations that the experience with semi-automation is a promising approach for continually speeding up the service creation process.

Keywords: Service-oriented computing; Service creation; Service composition; Pattern; Template

Background

The use of 'services' as the basic unit in the provision of access to shared IT systems and Internet-based applications is becoming more frequent, both because of the direct application of the concepts involved with the *service-oriented computing* (SOC) paradigm and because new and well-known applications are being made directly available over the Internet. One of the most interesting features of SOC is its ability to create new services out of existing ones, thus fostering the reuse of existing proven solutions.

However, service creation is currently a bottleneck when it comes to enabling a faster time to market for new and innovative services. These services will be made available to Internet and IT users, and this will allow companies to come up with new products, promotions, and offers, even to non-Internet users. The main reason for this is that most services are manually created, with little help from automated techniques, thus slowing down the rate at which innovation can be made available to a variety of highly demanding customers.

Most strategies proposed in the literature for dealing with this problem try to resolve the complex challenge of fully automated service creation, which is a very important open research challenge that still lacks a proper and operational solution. Existing solutions do not effectively clarify the many problems involved in fully automated service creation approaches because they concentrate on particular techniques and environments, rather than placing themselves as a part of a generic service creation architecture, that is, able to participate on the myriad of different technologies and methodologies present in service creation, execution, and management contexts. Fully automated techniques are currently hindered from being used in operational systems because, among other causes, they require services to have advanced semantic descriptions in addition to the basic interface. Since this is not currently required, service creators do not necessarily have the needed skills. Also, services used as examples by automated techniques presented in the literature are usually too simple, such as sending an SMS. Therefore, it is not clear whether the algorithms would work for more complex, and real, services.

It must be possible for non-experts in particular technologies to create new services relatively easily. In an ideal scenario, an entrepreneur with a computer, a connection to the Internet, a good, solid idea, and a credit card should be able to create a new Internet service and launch it automatically in any data center in the Internet cloud. The same reasoning applies to business analysts who work in companies that increasingly

* Correspondence: ernani@gprt.ufpe.br
[1]Federal University of Pernambuco, Professor Moraes Rêgo Av., 1235, Cidade Universitária, Recife 50670-901, Brazil
Full list of author information is available at the end of the article

depend on IT and telecom systems to provide services and products and to start new successful marketing campaigns.

The objective of this paper is to show that semi-automated service creation techniques are feasible, whereas fully automated ones are not a reality yet. Also, we advocate that increasing the level of automation may also increase the expressiveness and power of service creation tools. Service creation will always require some human intervention because ideas will always have to be expressed in a way that an automated tool will understand so that it can generate the needed code and configuration scripts. Therefore, we need to find out exactly what the minimum level of intervention required is, according to the knowledge we have at a particular point in time, and then continually add new levels of automation to the system. An interesting way of discovering this minimum level is by dealing with different approaches toward service creation with different automation levels. In other words, the introduction of automation should be made by evolution and not revolution.

We developed two versions of a *service creation tool* with different levels of automation based on an underlying framework called *service refinement cycle* (SRC), and both use the *service code* [1] to describe the final executable service. We found that by increasing the level of automation, we could significantly improve the functioning of the activities related to service creation, in terms of both human effort and the expressiveness of the solution.

The first approach, called *quasi-manual service creation*, has been developed with the strategic goals of shedding light on the problem and gaining insight from it, by exploring what we consider to be one of the most basic forms of automation that could be used to help the users with service creation. We call it 'quasi-manual' as a reference to this approach, wherein service design and building are completely up to the business and IT professionals. The second approach, called *pattern-based service creation*, is aimed at providing a more productive environment for translating a high-level service specification into an executable service code. Building on our experience with the quasi-manual approach, this approach adds up the existing knowledge about the use of patterns for system design and goes a step forward in the long spectrum line, from manual to automated service creation. The quasi-manual approach requires more manual effort to develop a new service, whereas the pattern-based one requires a more elaborate design and a more powerful tool, in order to help service creators.

The impacts of this work are manifold. Firstly, the experience of working with two different versions of a service creation tool so far confirmed our expectations that the experience with semi-automation is a promising approach for continually speeding up the service creation process. Secondly, our service specification proposal - the service code - plays an important role in the design of service creation frameworks, once one could guide the description of a service by referring to the final format defined in the service code. Thirdly, we make the point that service creation does not necessarily need bind the final solution to particular technologies since the beginning of the process, which provides flexibility and eases the deployment in many different environments.

The rest of the paper is structured as follows: some background and related work are presented, and the main underlying concepts of a service refinement cycle and of service code are presented. The methodology used for the development of this work is explained, and the concepts of service creation and a generic service creation tool are presented and the quasi-manual and pattern-based service creation approaches are introduced. Finally, the lessons learnt are discussed, where we pose some conclusions and present possibilities for future work in the lessons learnt section.

Service-oriented computing

Service-oriented computing is a paradigm that employs services as the basic unit for the separation of concerns in the development of computing systems [2]. For the purposes of this discussion, we adopt the definition of services [3] that describes it as 'a mechanism to enable access to one or more capabilities using a prescribed interface and which is exercised in a manner consistent with the constraints and policies specified by the service description'.

Service-oriented architecture (SOA) is an architectural model which advocates for the provisioning of functionalities in the form of services. SOA is believed to facilitate the growth of large-scale enterprise systems, allow Internet-scale provisioning and service usage, and reduce costs through cooperation between organizations [2], in that it can provide a simple, scalable, and interoperable paradigm for organizing large networks. *Web services* [4] are the most common technologies used to implement SOA services over the Internet.

Service composition

A service composition is a coordinated aggregate of services that have been assembled to provide the functionality required to automate a specific business task or process [2]. SOA advocates that any service is expected to be capable of participating as an effective composition member. This is shown in Figure 1, where the circles at the top represent atomic services and the polygons at

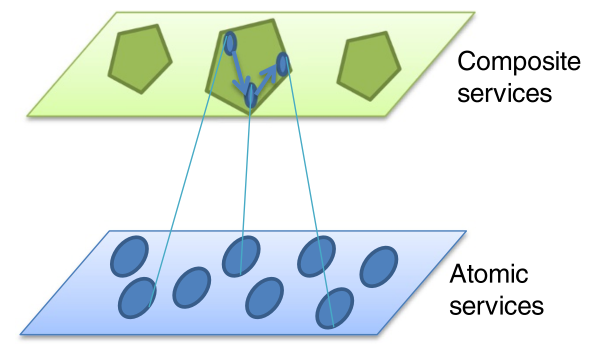


Figure 1 Atomic and composite services.

the bottom represent composite services made up by combining atomic services.

The composition of web services has been classified as *orchestration* and *choreography* [5]: two concepts related to the coordination or control of individual services that work together. Orchestration refers to the coordination of services from the point of view of one regent participant, who is able to control the flow needed for the process to be executed correctly. Choreography, oppositely, demands a certain protocol to ensure harmony and interoperability among the interacting participant services, without the need for a centralized control.

Policy refinement

Policy refinement is usually defined as the process of deriving low-level enforceable technology-oriented policies from high-level business-oriented goals or guidelines [6]. A typical methodology for policy refinement is the process of incremental decomposition, wherein at each refinement step (level), the generated policies are checked against the requirements of the policies from previous step for correctness and consistency. Although policies are intrinsically related to services and the problems of policy refinement and service composition are similar, proposals to address both problems together are not common.

Related work

The literature on service composition presents a considerable number of proposals [7], although most of them deal with the more challenging and complex problem of automated service composition, what relies on a number of intelligent techniques that have not proven feasible for all environments so far.

The proposal in [8] presents an approach based on automated planning, in which the composition is mapped to a plan and the existing services are used as its subtasks. The major setback in using this approach is the need of a detailed and correct semantic description of the services and their components. Because this is such a cumbersome and error-prone activity, the authors used parameter descriptions of the services (inputs-outputs) based on the data types extracted from WSDL descriptions. Since a full-blown service composition demands an impracticable computational complexity - especially in the cloud computing environment - a graphical user interface was designed to allow partial specification of the service, that is, developers are able to specify input, output, and possibly intermediate parameters and subservices through the tool and trigger the planning algorithm to obtain a composition that leads from the inputs to the desired outputs.

Another proposal for automation of service creation is presented by [9]. It describes an architecture-based approach for the creation of services and their subsequent integration with service-requesting client applications. Their technique uses an architecture description language to properly describe the services and achieve the runtime integration using middleware technologies. In order to obtain deployable software, the developer implements a client concerning the services that it must invoke to form an application. All other necessary codes, including the code necessary to realize the connections between the client and employed services, are generated based on the specifications describing clients, services, and connectors.

We believe that a continuous progress of semi-automated techniques is much more useful for understanding the real problems and trade-offs in this area than attempting a fully automated approach, ignoring the many details that will be left unsolved along the path, such as the necessity to address many existing standards and technologies. With the continuous automation approach, we expect to achieve the final goal of fully automated techniques sometime in the future. Automated service composition techniques sometimes recur in what we call 'mystic boxes' in their architectures because they are frequently poorly described, making it difficult to understand what exactly has been done and to evaluate their results.

Service composition may happen offline or online, i.e., at creation or execution time. However, this very important difference is not always made clear in proposals. Agarwal et al. [10] present some approaches for dealing with the multifaceted problem of *web service composition and execution*, defining it as the process of creating workflows which realize the functionality of a new service and their subsequent deployment and execution in a runtime environment. Our work acknowledges the difference between service creation and service execution, and the approaches for semi-automated service creation presented here are mostly targeted to work offline, even though the actual subservices to be executed are only selected at execution time due to the late binding feature.

The authors of [11] are motivated by the business need to quickly develop and deploy solutions, in order to reduce the time to market for telecom services. They address the use of templates to skip the repeated development phases of traditional creation techniques. In order to become a new service template, a workflow must be carefully analyzed so that its use can be generalized. If approved for that purpose, the general behavior of the service must be extracted and the specific parts of the workflow must be marked as *service configurable points*. Like in our proposal, generic workflows may be instantiated for different solutions according to parameters specific to each case. Still, this approach does not make clear the role division between business and IT personnel, whereas our approach does consider different levels of templates of patterns and delegates different responsibilities in the service creation process. We advocate that this separation of activities is important to leverage the flexibility of service creation (i.e., allow the creation in various environments) and that a well-defined role assignment helps to understand each part of the creation and eases the automation of these specific activities.

At last, motivated by the promising low cost and agility in software development glimpsed by automated service creation concept, the authors of [12] gather the results of a long-term observation on techniques to accelerate service description and deployment. In the preliminary background knowledge presented, the paper points out two composition categories (manual and automated), ignoring the benefits of a midterm approach, namely the semi-automated creation. They also refer to many commonly used standards and underlying technologies, all related to service-oriented computing and architecture. After listing the used criteria of their research on service definition automation, such as description, matchmaking, combination, and selection, a summarization of the state of the art on this matter is presented, along with the future challenges and directions considered by the authors.

Case study

The necessity of the proposed creation approaches is illustrated, and nowadays' weaknesses in offering a new service is pointed out: suppose an entrepreneur detects the lack of a DVD rental store home delivery service, though there are available online DVD reservation and delivery services whose owner companies provide accessible interfacing for their services. In order to maximize the profits and reduce time to market, the entrepreneur should be able to create and deploy the idea with least necessary interaction to other people.

In the manual creation, all the mentioned services must be individually designed and implemented. The entrepreneur must hire a developer, whose technical expertise will permit the successful integration of all subservices. At the design phase, a feasibility study of the technologies and standards must be addressed, that is, programming languages and web technologies compatible with the existent solutions. Hereinafter, a number of specific *integrated development environments* (IDEs) are used to implement the final application in a technical (programming) language, which comprises calls to the existing DVD rental service, delivery service, and billing service. The front-end design and the hosting for the final service are other concerns.

We advocate that a company which follows one of the approaches presented in this paper may offer service creation and hosting facility for this environment, which in its turn pans by a minimal set of creation-involved people. Possible embodiments of this scenario are presented further in this text to illustrate the use of the proposed creation approaches.

Service refinement cycle and service code

Service creation is all about the refinement of a high-level business idea into running code. A key part of our proposal is the concept of service refinement, a combination of service composition, and policy refinement which involves refining a higher level service specification into an integrated composite of lower level services. Two important underlying concepts in our proposal are the service refinement cycle and the service code.

The service refinement cycle

SRC [1], which is a result of the service refinement approach, is an integrated abstract framework for dealing with the service life cycle, which defines a common 'language' (or structure) for specifying and understanding service composition. It is composed of a sequence of phases (service, process, policy, and binding), as depicted in Figure 2, where each phase adds some features for enhancing the capacity of services to adapt their behavior, according to different requirements. Transitions represent

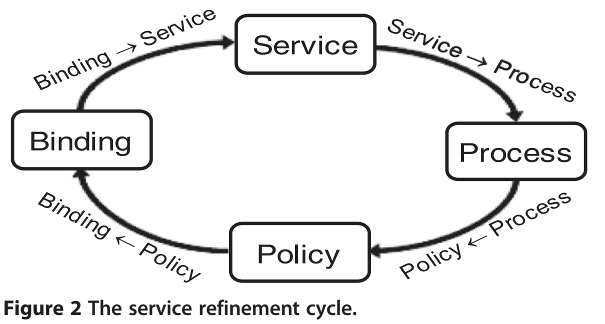


Figure 2 The service refinement cycle.

the way in which an execution engine passes from one phase to the other.

Phases contain the information that defines services and that is generated during service creation, when the user is assisted by an appropriate tool. The objective of and reason for each phase is as follows:

- *Service*. This contains a standardized service description following the SOA paradigm which assumes that services must be widely advertised.
- *Process*. Complex services are usually defined as being composed of other services, which always follows a process. The process phase is composed of one or more abstract tasks (not real services yet), which interact with each other in certain ways.
- *Policy*. The role of policies in service refinement is to provide flexibility to both the composition of services and to the resulting ability to adapt behavior.
- *Binding*. The binding phase isolates the internal phases (service, process, and policy) from the outside world. It maps internal abstract services to external real ones.

Transitions specify links between phases which are designed during service creation and followed by an engine at execution time. Internal transitions link phases of the same service entity, whereas external transitions link different services. The transitions between internal phases are dynamically bound by default, which means that one phase sends a request to the next phase, to be evaluated at runtime. Since the structure of the refinement cycle is flexible, each phase within a particular refinement chain contains a pointer to the next phase. For each application programming interface (API) function of the service phase, there must be an associated logical refinement. However, a single incoming external request received by the service phase may cause many outgoing external requests to be issued by the binding phase.

Only the service phase is mandatory for the refinement of a given service API function. The use of the process, policy, and binding phases is optional, which means that there may be transitions that skip certain phases. Whether or not to use a particular phase in a refinement is up to the service designer and may also be modified by the system administrator at runtime.

The main reason for dividing service refinement into four phases is to provide separation of concerns. It creates independence between phases when designing a new service. Service description, process, and policies may even be created by different people or by different intelligent systems, regardless of the actual services that will implement the external functions. By dividing a service refinement into pieces, we can expect service creation (manual or automated) to become easier.

In Figure 2, the service refinement cycle is represented as an iterative process, where the subservices that make up a particular composite service may in turn be refined into a composition of other services, and so on. The interaction between different subservices builds, at runtime, multiple layers (of services) that participate in the implementation of a single, higher level service.

The service code

The main realization of the SRC is the service code, which contains executable specifications (code) for the four phases of the cycle. In other words, the service code is the outcome of a service developed according to the SRC. If the content of the service code is packed into a standard file (similar to a .jar file), any compatible engine should be able to execute it.

Service creation and execution

Three important stages of the service life cycle that are supported by our framework for service refinement are service creation, execution, and management. The interface between those stages is the service code, which is generated by a service creation tool and deployed to be run by a service execution engine. Given that the SRC defines the specification of services as the service code, automation of service execution should be possible by a generic engine. A service management tool has access to this code and is able to update it on the fly according to demand.

Methods

The service creation approaches we developed are based on the service refinement cycle, which has been designed as part of a project involving the concepts of service composition and policy refinement. The particular work described in this paper consists of three steps. The first step is the development of a generic service creation tool (SCT), i.e., the platform that supports various service creation approaches. The second and third steps involved are, respectively, the design of the quasi-manual and pattern-based service creation approaches and the development of the associated tools.

Since our goal in proposing automation in service creation is to make it more accessible to a non-technical user, for prototyping, we aimed at eliminating the need of development environments, making the tools available according to the cloud computing paradigm. As far as software development is concerned, our tools are based on the Web as a platform paradigm in order to be able to easily cope with the *software as a service* (SaaS) concept. The usability issues usually associated with web-based applications have been overcome by the use of state-of-the-art technology, thus avoiding noticeable negative impacts for users, when compared to standalone tools. Our tools use Asynchronous JavaScript and XML (AJAX) [13] because (a) it fulfills the required development agility, (b) it is able to create applications with high levels of usability from the user point of view, and (c) it has been widely adopted by the web development community. Our tools for service creation have similar roles, so common architecture and development technologies were used to implement them. For example, we developed a common XML parsing and generation module and a shared information model. In AJAX, client-side modules of the application hold the widgets that will be executed on the browser, while a server-side module provides some advanced features. Therefore, all user interactions that use the service creation tools happen through the client-side application, whereas complex operations such as parsing, generation, and remote communication are done on the server side.

Google Widget Toolkit (GWT) [14] is a framework that is publicly available and has been widely adopted for fast AJAX-based development because it reduces the complexity of dealing with client and server sides, remote invocation, and diagramming of graphical components for the current variety of browsers. Developers deal with a simple Java library to invoke a few routines and are mainly concerned with program logic, while GWT handles the translation and deployment of the application. We used a GWT plug-in for the Eclipse IDE [15] due to its smooth integration and testing facilities.

We also developed a *service execution environment* which can be used to run the service code created by our tools, but the description of this tool is outside of the scope of this paper.

Service creation tool

Service creation is the process of building the service code, given a high-level specification of a service and the availability of particular subservices for implementing the process. It is a sequence of steps that start from the initial conception of a new service up to its deployment.

Service creation approaches

A challenge is how to fill in the refinement cycle phases with the content that is necessary for specifying a service. An important issue raised by this approach is that the refinement cycle is orthogonal to service creation approaches. In other words, as long as the relationship among its four phases is observed and represented in the service code, the particularities of each approach do not interfere in the final result. The importance of this concept is that it opens up new research opportunities and a clearer view of the area because it allows different methodologies to be compared under a single framework, which is not common in this area.

Currently, we envision at least three general approaches for service creation:

- *Manual service creation*. This occurs when service design is done completely by a human operator, who may or may not use a tool for assisting him/her with the generation of the service code.
- *Fully automated service creation*. An intelligent technique, e.g., based on reasoning, can automatically find optimal services, processes, policies, and bindings for fulfilling a higher level service description. This is currently an interesting open research problem [7], although it is not yet operational given its higher complexity.
- *Semi-automated service creation*. Along the line between the extremes of exclusively manual or automated creation, there may be a myriad of semi-automated techniques. This means that the service designer might be assisted by a tool that provides different levels of automation for particular activities but still requires some level of manual intervention.

The two proposals for service creation presented in this paper follow the semi-automated approach for service creation, with an increasing level of automation. We found out that with a little increase in the computational skills of the users, a corresponding increase in the expressiveness and power of the technique (and tool) may be obtained.

Architecture of a generic service creation tool

The activity of service creation is supposed to be assisted by an SCT, which should follow two basic guidelines. The first guideline states that services should be easily created by non-experts in a particular technology. In an ideal scenario, an entrepreneur with a computer, a connection to the Internet, a good solid idea, and a credit card should be able to create a new Internet service and execute it automatically in any data center on the Internet. That is to say, a service creation service and a service execution service should be available for entrepreneurs using cloud computing concepts [16], specifically SaaS and IaaS technologies [17]. The same reasoning is applicable for service creation within an organization, where both service creation and execution services are supposed to be available only via intranet.

Unfortunately, technology is not yet ready for such a level of automation in service creation and execution. Also, not all services are created equal, and frequently, they are not simple enough for operators without sound technical skills. Currently, it requires those professionals to interact with complex concepts such as business process modeling languages or even general purpose programming languages. Therefore, the second guideline states that business and IT experts should be assisted by a semi-automated tool to speed up service creation and consequently time to market.

Figure 3 depicts how an SCT fits into such a scenario and outlines the role of the professionals involved in creating a new service. The outcome of the service creation activity is the service code as well as a service graphical user interface (GUI), which invokes the features of the service code. Both are executed by a service execution environment that may be located anywhere in a data center in the Internet cloud.

As shown in Figure 3, service creation is divided into three different modules or subservices which are operated by people with different skill sets. The *service designer* allows an entrepreneur or a business analyst from an organization to specify a new service in an abstract way that more closely resembles the business language. This high-level service specification may be composed of, among other information, a service description, a service API, a set of business policies, and eventually, hints for business processes. The latter is provided as a feature of a particular service creation tool (such as the one that follows our quasi-manual one) that allows the service designer to provide a list of subservices to be used in the refinement of a particular higher level service. The output of the service designer is an abstract service specification, called *service guideline*, expressed in an ontology-based language such as Web Ontology Language (OWL).

Service builder is a tool to be used by a system developer (an IT expert) and contributes to the development of the service code from the service guideline, coming from the service designer. For any given service guideline, many different versions of service builders may exist, with different levels of automation and which use different techniques and approaches. As a matter of fact, we have developed two versions of it, increasing the automation level with each version. Nevertheless, since its purpose in service creation is fully defined, service designer is expected to be preserved in a generic SCT, as long as the service guideline description it generates is able to express meaningful services and works well with different embodiments of the service builder.

We have developed two variations of semi-automated service builders using the architecture presented in Figure 3. An interesting side effect of our developments has been a generic platform that may be used to design, implement, test, and compare different strategies for semi-automated service creation. Given a service guideline, different strategies may be used to build the resulting service code, which may be compared using qualitative and quantitative approaches.

In spite of the paramount importance of creating the service code, its aim is not to be directly accessed by service users but rather to be accessed by programs via an API. Therefore, in any practical scenario for creating and executing services in the future Internet, a GUI should be made available for invoking service features. The *front-end designer* is a tool used by a *GUI designer* that transforms user ideas into calls to the service API. It makes the transition from the user to the service realms. However, despite of the importance of the front-end designer, it is out of the scope of this paper, since the GUI is orthogonal to the service itself and different types of GUI may be used, such as versions for the Web and for smartphones.

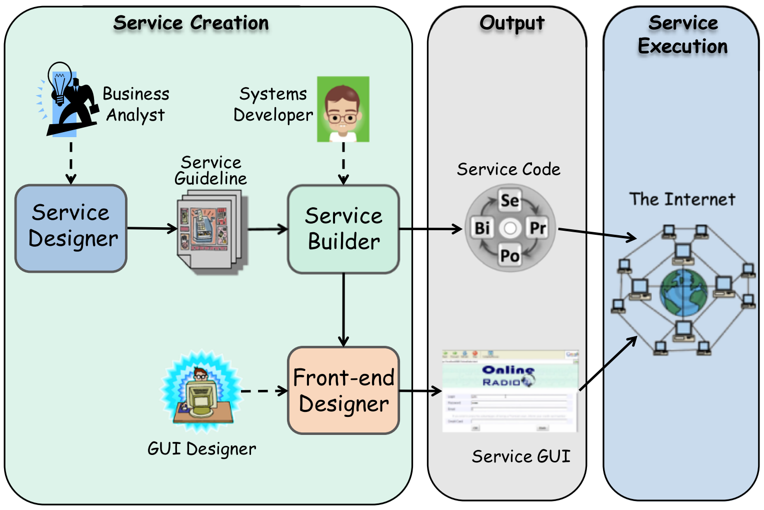


Figure 3 A generic architecture for a service creation tool.

Quasi-manual service creation

The first version of our SCT has been developed with the strategic goal of shedding light onto the challenge of service creation and gaining insights from it, by exploring what we consider to be one of the most basic forms of automation used to help users with the process of service creation. We call it *quasi-manual* as a reference to this approach, where service design and building is completely up to the business and IT professionals. As a matter of fact, during the design and implementation of this technique, concepts such as the service guideline emerged and shaped the particular format and content of the service code, thus leveraging the development of more advanced representation forms as well as new and innovative ways to deal with the service creation problem.

Overview

In order to provide flexibility for service creation, the quasi-manual approach is divided into two phases, which follow the generic architecture (Figure 3). As depicted in Figure 4, the design phase is aimed at capturing the idea for a new service of the business analyst or entrepreneur and to transform it into a service guideline. Next, in the building phase, the system developer who is using the service guideline, as well as his/her technical skills, is responsible for assembling the necessary technologies for the realization of the service and the generation of the final code that is to be deployed. As the picture implies, service designer and service builder are distinct software applications to be used by people in different professional roles with complementary skills, although a single person may play both roles in some organizations. The job of the *analyst* and the *developer* is mostly manual but includes some assistance, thus explaining the name quasi-manual.

Outcomes: service guideline and service code

Service guideline is a description of the service behavior in natural language, though it is written as an ontology, and should represent the expectations of the business manager, including features and constraints. Its main purpose is to organize the service description into well-defined sections to allow the developer to implement the solution using a formatted 'requirements document', which is less prone to presenting inconsistencies. If the creation of a service is seen as a project, the guidelines may be considered a requirements document. In addition, service guidelines also encourage reuse as they are designed for a particular purpose in a particular context but can be used elsewhere and can even be implemented in different ways.

The description of the service guideline, presented in Table 1, contains some attributes of SOA, as proposed by the Open Group SOA Ontology. Important attributes of the service guideline are functions, policies, inputs/outputs, and processes. A service is composed of one or more service functions, and each one is responsible for a particular operation performed by the service. Functions may be invoked individually from a service API, whereas a service may be considered a wrapper used to group the functionalities provided by its functions. Policies define constraints that may be applied to an entire service or to particular functions only. As functions are the parts of a service that are actually invoked and executed, criteria to access them are needed and should be represented as input and output parameters. Process definition is needed in the service guideline for the quasi-manual approach since there is no automation in the building phase for composing multiple services together. However, in a more automated solution, it may be changed by altering the 'pre' and 'post' conditions of the service functions.

Figure 5 depicts a possible structure of a sample service guideline for a delivery function in an online bookstore service. Important components of a service description are the function policies, such as P1 and P2, the parameter, and returning types of the functions, the function process.

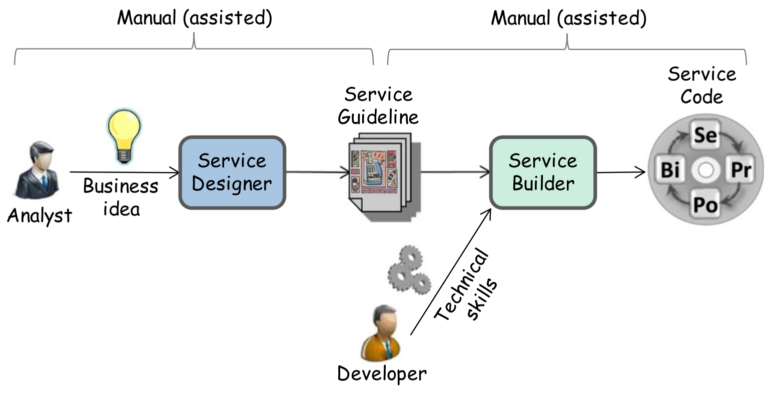


Figure 4 Activity flow of the quasi-manual approach.

Table 1 Attributes of a service guideline description

Attribute name	Description
Service ID	An identifier for the service being worked on
Service description	Natural language description to help the actors involved in the service development
Service owner	The domain of the service to help on legal matters
Service version	Internal version control
Service policies	Definition of the constraints that apply to all components of a service. It is composed of a name and a description in natural language
Function ID	The identification of a functional part of a service, i.e., a function that may be invoked from the service API, always associated with a service ID
Function description	A description in natural language of how the function should behave to help developers during the service building
Function version	Internal version control of each functional part
Function inputs and outputs	Input and output parameters of service functions represented as a pair name-type
Function policies	The same as service policies but specific to individual functions
Function process	Describes the steps of a workflow in natural language, so the developer can translate it to service code more easily

Service code is a concept, so it does not depend on any particular technology. However, technologies for the four phases of the service refinement cycle must be specified to create a running system. The combination of all standards and technologies used to define a service must be easily understood by an execution engine. As the service refinement cycle claims, services for cloud computing and SaaS paradigm should, in general, be flexible and portable enough to be enforced even when using mixed technologies. In this context, the concept of service code was created to name the set of documents defined in computer language technologies that together describe a service in a back-end level. Some options used for the quasi-manual approach are as follows:

- *Service*. Service description is specified using the Open Group SOA ontology [18] based on OWL [19].
- *Process*. Web Service-Business Process Execution Language (WS-BPEL) [20] is used to specify our processes. Alternatively, an ontology language may be used, such as OWL services (OWL-S) [21].
- *Policy*. Policies are written in an extended version of Extensible Access Control Markup Language (XACML) [22], but they may be also expressed using an ontology language like Semantic Web Rule Language (SWRL) [23].
- *Binding*. The binding phase is also specified in OWL.

The Open Group SOA Ontology states the information model to pave the building of valid service-oriented approaches. WS-BPEL is the reference for process orchestration in web-based environments, and XACML has a complete framework to handle access control entitlement, extended by [24] to deal with more classes of policies. SWRL is a project in ongoing evolution, targeted at adding more information to ontology descriptions and leveraging semantic reasoning.

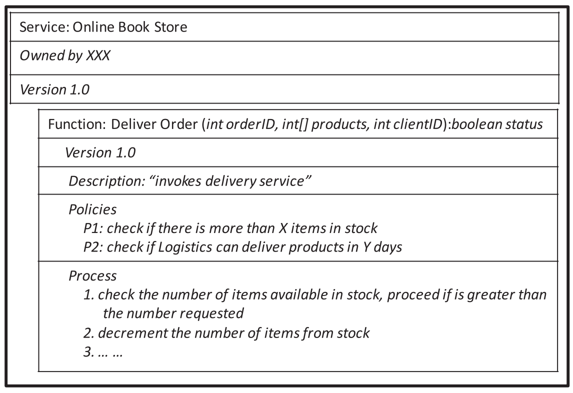

Service: Online Book Store
Owned by XXX
Version 1.0
Function: Deliver Order (*int orderID, int[] products, int clientID*):*boolean status*
Version 1.0
Description: "invokes delivery service"
Policies
P1: check if there is more than X items in stock
P2: check if Logistics can deliver products in Y days
Process
1. check the number of items available in stock, proceed if is greater than the number requested
2. decrement the number of items from stock
3. … …

Figure 5 Service guideline example for a single function.

Modules: service designer and service builder

We developed two tools for the quasi-manual approach: service designer and service builder. The service designer is used for service specification, which results in the service guideline, which is the formatted description in natural language of the service and its components. It consists of a web GUI, where the business user can input the component parts of the guideline. The most advanced GUI feature of this tool is the capability provided to the user to define the abstract natural language workflow of the processes via an intuitive drag-and-drop workflow assembling component. The mastermind of the service, referred to here as the analyst, uses the service designer tool to write the expected behavior of the functions, interfaces, and constraints. Once the service description work is finished, the service guideline is ready to be possibly shared and addressed by a variety of different service building tools, although for the quasi-manual approach, we developed one particular tool for fulfilling our purposes.

The service builder, which is responsible for generating the service code, works as an 'aggregation' of back-end technology editors since it is able to translate the abstract description into many executable standards which will comprise the service code. Its development gave special attention to the fact that technology descriptors for the whole service specification must be provided, according to the *Reference Model for Service Oriented Architecture* [25]. Via the service builder, the IT professional called developer is able to implement the guidelines according to a few predetermined standards for workflow definition and entitlement specification, that is, the tool is able to translate the service specification to specific back-end executable technologies, such as BPEL + XACML, XPDL [26] + Ponder [27], and Java + Rei [28]. In other words, a specific instance of the tool has to be used according to its desired output. This is where the general architecture for the SCT plays a very important role. It is aimed at providing flexibility by relying on the fact that a guideline that is specified in natural language can be accomplished by more than one different building tool.

Case study: DVD delivery

In order to obtain results about the aforementioned scenario illustration using the proposal in this section, John, the entrepreneur, should describe the business idea, the 'DVD Delivery', using a web platform of service designer in order to obtain a proper service guideline, though, to make the solution available as a service, he should use a provider infrastructure. For that sake, John contacts Paul, a system developer, who is able to develop a fully working implemented version of John's guideline. Paul will use the specific service builder which generates service code for the provisioning plan chosen by John (the customer of the service provider).

A snapshot of this scenario is showed in Figure 6 where the chosen technologies for service code are WS-BPEL for workflows, XACML for policy entitling, and SQL for data storage. Optionally, builders that would generate service code for different technologies could be used, as Java for workflows, Ponder for policy entitling, and LDAP for storage.

In the first moment, from the developer's point of view, this approach reduces development time and complexity because, instead of being concerned with the complete creation process, expert users split the implementation into smaller problems, each of which inside the context of its own expertise. Using the same assumption, we also make the point that activities developed by both John and Paul can have a significant improvement, which is avoiding repetitive tasks. This insight, which resulted from the observation of this minor modification in service description, leads us to rely on a known software engineering artifact, aimed at accelerating specification time and becoming less error-prone design patterns.

Pattern-based service creation

The pattern-based semi-automated service creation proposal is aimed at providing a more productive environment for translating a high-level service guideline into an executable service code. Building on our experience with the quasi-manual approach, this approach adds up the existing knowledge the use of patterns for system design and takes a step forward in the long spectrum line from manual to automated service creation.

Overview

When a new service proposal is conceived of by its creator (the analyst), the process usually starts as soon as the service requirements are defined and the needed resources are gathered. After that, the implementation begins according to the methodologies established by the organization. The pattern-based service creation approach is an attempt to both provide a higher level of assistance to the developer during service building and to develop services without the need to bind the solutions to specific technologies, at first. This we learned from our experience with the quasi-manual approach, and we believe that service descriptions should contain the main components which consistently define their behavior, constraints, and interfaces and should be dependent on the system onto which they are going to be executed. The translation of the service code into the languages and standards that will actually be enforced should happen in the very last moment, immediately before deployment.

The pattern-based approach involves describing the service and its components in a way that fosters reutilization of previously developed services. Three levels of information representation provide the maximum balance of simplicity and flexibility, and each of them is assisted by a pattern/template used by a corresponding tool. Service guidelines are similar to the quasi-manual approach presented in Table 1. *Abstract service code* is all an ontology-based and technology-independent version of the service

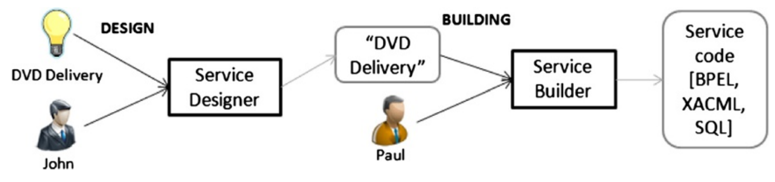


Figure 6 Use flow of quasi-manual proposal to develop DVD delivery service.

code presented in the 'Outcomes: service guideline and service code' subsection. Finally, the *executable service code* is the concrete description of the service ready to be deployed, i.e., the very service code presented in the 'Outcomes: service guideline and service code' subsection.

In order to speed up the generation of such description files, this approach leverages the use of patterns and templates to guide the first steps of development. Design patterns guide the creation of a formal requirements document (the service guideline) which, in turn, can be translated into a technology-free consistent description (abstract service code) assisted by implementation patterns. The abstract service code is finally compiled into the executable service code, with the help of technology templates. In the pattern-based approach, we added a third phase in the service creation process, which is aimed at the compilation of the service to generate the final service code.

Each tool used in this approach may generate a new outcome, in addition to the service representation forms, which are patterns and templates. The service designer uses design patterns together with the business idea to generate a guideline. The service builder uses *implementation patterns* to assist with the process of transforming the guidelines into actual descriptions, not yet efficiently executable. The service compiler generates executable service code using the output of the builder, using templates to translate the generic description into concrete technologies that are ready to be deployed. The actors responsible for designing, building, and compiling phases are respectively the analyst, the developer, and the administrator of the service provider, and they are allowed to deploy the solution in an execution environment, as shown in Figure 7.

By doing this, we expect to provide flexibility to the service descriptions because the outputs will be used by different professionals and organizations, in addition to being specified in a self-contained fashion. Compared to the quasi-manual approach, the level of automation is greater in the pattern-based approach due to the use of patterns and templates as well as to the reutilization of previous efforts in service creation. Figure 7 shows that the service builder provides much more assistance to the user and therefore is considered a semi-automated tool and that the service compiler is a fully automated tool. The service designer also provides more assistance to the user, but it may be considered a manual tool since its activity is highly human-dependent.

Outcomes

The outcomes (intermediate and final) of the pattern-based approach are the service guideline, the abstract service code, and the executable service code. The service guideline for the pattern-based approach is similar to that of the quasi-manual approach, as presented in the 'Outcomes: service guideline and service code' subsection.

The abstract service code (ASC) is the bundle of descriptions that correspond to the machine-like specifications of the service components. The ASC is the output of the service builder, built from the service guidelines as well as the input for the service compiler. It is 'abstract' because its content is not tied to any technology and yet is considered 'service code' because it is representative and generic enough to be translated into any computer language. The foundations of the abstract service code are based on building blocks that represent every part of a service description in a syntax that is supposed to be easily translated into programming language grammars. This feature is useful for the compiling phase of the pattern-based approach. A possible instantiation of the abstract function deliverOrder is depicted in Figure 8, where some components of implementation described in the service guidelines, like the attribute names and workflows, are now specified in a computer-interpretable manner. That is to say, a processing of the information can be done in order to obtain a more specific description. Although the language used in the picture does not correspond to any existent

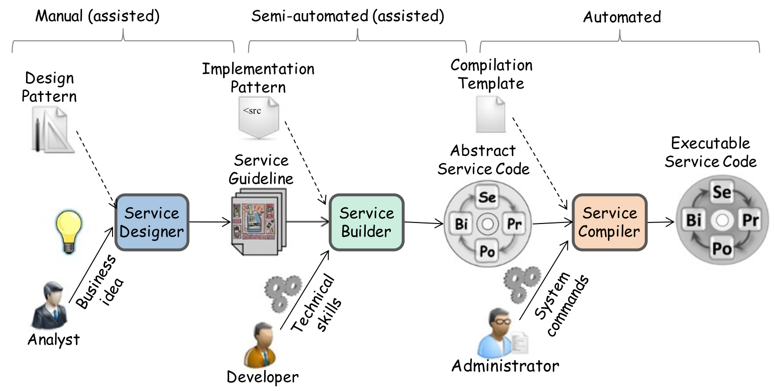


Figure 7 Activity flow of the pattern-based service creation approach.

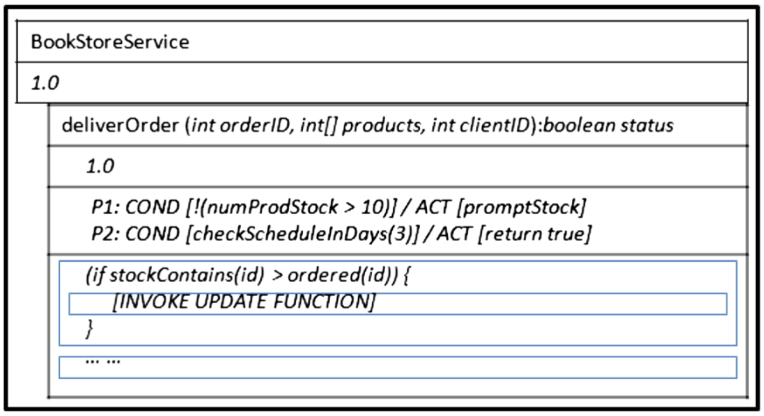

BookStoreService
1.0
deliverOrder (int orderID, int[] products, int clientID):boolean status
1.0
P1: COND [!(numProdStock > 10)] / ACT [promptStock]
P2: COND [checkScheduleInDays(3)] / ACT [return true]
(if stockContains(id) > ordered(id)) {
[INVOKE UPDATE FUNCTION]
}
... ...

Figure 8 Example of the abstract service code for the deliverOrder function.

technology, we suggest the use of ontologies to describe the abstract service code in order to take advantage of its semantic features. Still in the figure, policies (P1 and P2) are now represented as condition-action pairs, and a snippet of a programming code can be seen in the workflow description.

The abstract service code is supposed to achieve an excellent balance between design, implementation, and execution since it combines the intentions of the service design with the particularities of the implementation while also taking into consideration the singularities of enforcement. The ASC is more implementation-driven than the service guideline but is not tied to any particular technology like the executable service code is. Therefore, it is the perfect stage of the service representation at which to add semantic behavior, which would not be straightforward in the other two stages. In addition, this capability may add a higher level of automation to the system since a proper execution engine can solve building time issues, like the lack of appropriate components, or can suggest more appropriate ones in certain cases. The ASC can be also thought of as an intermediate code, similar to the Java byte code in that it must be translated to a particular underlying technology to be executed in a target computing platform.

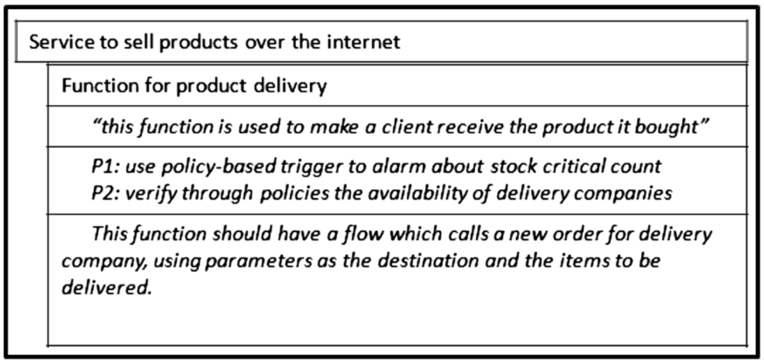
A service creation tool must be prepared to generate an executable code, i.e., files whose structure can be interpreted and enforced by some execution engine. Service guidelines are descriptions for business people, and the ASC is a format targeted at representing the service implementation in a computer-like ontology-based language. The final output of the framework must be able to be deployed in an execution environment, as stated by the service refinement cycle. The executable service code (ESC) is the equivalent of the service code in the quasi-manual approach. It is a package of files generated by the service compiler, allowed to be deployed, and executed in real systems.

The use of patterns and Templates

A design pattern is usually defined as a proven solution to a commonly occurring problem which is well documented and part of a collection of similar solutions [29]. Although it is not a finished design that can be transformed directly onto useful artifacts, it contains an abstract description that points out how to solve a problem, even in different application domains.

Inspired by this view, design patterns for service creation are used to support the definition of services in the topmost business level, i.e., they act as a reference for the definition of service guidelines. In other words, design patterns for the service design stage of service creation assist business analysts by providing advice on how to solve a problem by referring to proven working solutions. Design patterns in our proposal come as a natural language description of the requirements for generating service guidelines. They will assist the business level part of service creation, taking advantage of the layered nature of the framework; that is to say, design patterns may also be specialized to compose functions that have been refined differently yet are similar in behavior. An illustration of how design patterns look is shown in Figure 9. Similar to a service guideline, in this picture, one can notice tips and hints for defining policies (P1 and P2) as well as a high-level description of how the workflow should behave.

Design patterns represent a useful guide for the stage of service creation that is performed by the business analyst, though it leaves the actual ‘coding’ of the service still unassisted. Therefore, we use implementation patterns to speed up the work of the system developer during the service building stage. Implementation patterns are specializations of design patterns, and one will notice that the former are closer to source code syntax and must be dealt with by system developers, whereas the latter are business level specifications and must be dealt with by business personnel.



Service to sell products over the internet

Function for product delivery

"this function is used to make a client receive the product it bought"

P1: use policy-based trigger to alarm about stock critical count
P2: verify through policies the availability of delivery companies

This function should have a flow which calls a new order for delivery company, using parameters as the destination and the items to be delivered.

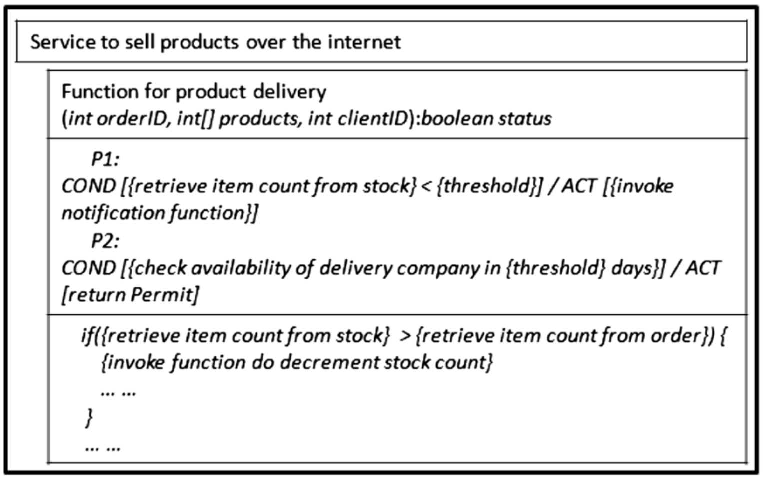
Figure 9 **Example of a design pattern for selling products on the Internet.**

Implementation patterns must define the framework for all further development, including constraints, interfacing, and even the operation execution flow. Alternatively, implementation patterns may be also considered templates for the abstract service code with blanks that must be filled in on a case by case basis, according to the particularities of the service being created. For instance, if the process that implements a service function points out that in a certain step a policy engine should be invoked, the blank for policy enforcement - in the instance of the implementation pattern - should be developed by each organization. Figure 10 depicts an example of an abstract service code with empty blanks for describing an implementation pattern.

As mentioned, in the last stage of service creation, it is mandatory to translate all service description data to a standard that is allowed to run on execution engines. In most current deployments, a service specification is manually implemented before becoming part of an execution environment. Our purpose is to skip the manual coding at the programming language level by providing compiling templates to service developers. Compiling templates take advantage of the building blocks of the ASC to perform mappings from their abstract representation to their respective counterparts in the computer language of the ESC. Every structure of a particular ASC instance must be converted to one or more computer language structures that can consistently perform the designed behavior. Compiling templates represent the most visible automation level in the pattern-based service creation approach since they completely eliminate computer language writing by hand. The service compiler deals with templates in order to consistently translate its inputs into specific computer languages.

Module: service designer, service builder, and service compiler

As shown in Figure 7, the service designer is the first tool used in the service creation process by a business analyst for defining the general guidelines of a service. The service designer does not force the use of patterns; they are optional because a helpful pattern will not always be available. The analyst may choose to create a



Service to sell products over the internet

Function for product delivery
(*int orderID, int[] products, int clientID*):*boolean status*

```
    P1:
COND [{retrieve item count from stock} < {threshold}] / ACT [{invoke
notification function}]
    P2:
COND [{check availability of delivery company in {threshold} days}] / ACT
[return Permit]
```

```
  if({retrieve item count from stock}  > {retrieve item count from order}) {
     {invoke function do decrement stock count}
     ... ...
  }
  ... ...
```

Figure 10 **Example of an implementation pattern: a mix of computer language and natural description.**

service from scratch, which may produce another useful pattern for future usage. Therefore, the service compiler also plays the role of a design pattern creation tool, as shown in Figure 11. Once design patterns follow the same rules used for service guidelines, the service designer can cause both artifacts to fully conform with one another. This tool includes simple editing features that permit the shaping of a requirements document so that it conforms with the service guideline specification. As a tool targeted at non-technical users, like entrepreneurs and business analysts, the service designer lets them remain unaware that they are designing a standardized requirements document.

The service builder is the tool used to generate the abstract service code. In addition, similarly to the way that design patterns are generated by the service designer, implementation patterns may be generated by the service builder so that advanced users can create new implementation patterns that comply with a given design pattern. The creation of implementation patterns with the same tool that uses them as inputs aims to provide better conformance between the implementation patterns and the abstract service code.

Figure 12 depicts the inputs and outputs of the service builder. The system developer takes a service guideline as the main input for generating the ASC as the main output and uses his/her technical skills to choose an appropriate implementation template and to fill the blanks with the other specific information. As a by-product, a new implementation pattern may be generated.

The service builder is a definition tool whose output is an abstract definition of a service written in a computer-like language. Some service components - especially process flow and policy description - must have a specification syntax whose dictionary embraces all possible structures. For a process definition, flow control entities must be part of the syntax as conditional deviations,

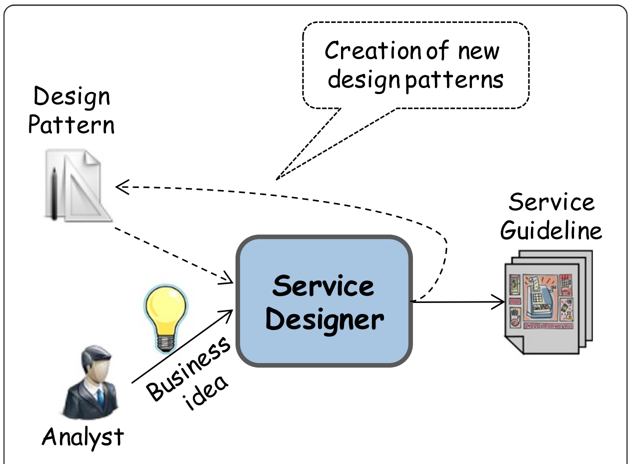


Figure 11 Inputs and outputs of the service designer.

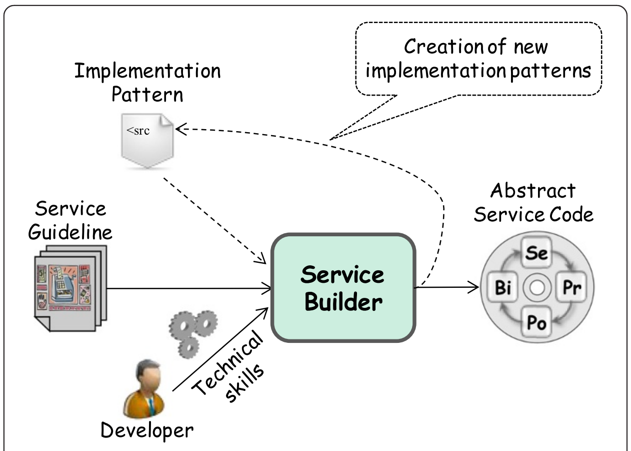


Figure 12 Inputs and outputs of the service builder.

loop occurrences, and external invocations in such a way that the developer is enabled to define every possible execution flow. The same reasoning can be extended to apply to policies. A policy, as defined in the literature, is composed of mandatory parts that demand a language with specific syntax and semantics.

Advanced versions of the service builders may provide many features for defining processes and policies, for instance, whereas simpler versions must provide at least the most basic structures. The latter will demand more effort from the system developer but will allow any flow to be defined. Our service builder belongs in the second class, meaning that it is a simple tool. The service builder is considered the main tool of the pattern-based approach since it provides higher levels of automation than the quasi-manual approach.

The service compiler is the tool that translates the abstract service code into an executable service code for a particular target service execution engine. The translation from the ASC into the ESC, using compilation templates, is allowed because the ASC syntax is fragmented, meaning that the attributes and values of each fragment can be read by the service compiler and translated into a computer language. The specific computer language that has been chosen as the standard for template translation may have implementation issues related to scope declarations and support files, which are not easily resolved by a simple translator.

Unlike the service designer and builder, where the use of patterns is optional, for the service compiler, the use of compilation templates is mandatory because it is a fully automated tool. The compilation templates are dictionaries that keep record of how each part of the ASC syntax should be addressed by the compiler in the corresponding executable language. Templates follow a limited yet complete syntax that refers to every possible part of policies, assignments, and processes.

Summarizing, all tokens such as conditionals, loops, operators, and literals in the ASC syntax must have an equivalent in the technology-specific language of the ESC in order for the compiling process to succeed. The service compilation process is based on Simple Transformer (SiTra), a model transformation framework targeted at creating prototypes, which has been used for the translation of OWL-S into BPEL [30].

The service compiler also has the ability to generate new compilation templates, as depicted in Figure 13. The system administrator configures the compiler, which does not need human intervention to generate the executable service code, using a compilation template.

Implementation issues

A service specification, as described here, involves a set of functions with a certain level of independence from each other and from the service they belong to. This resemblance to the object-oriented (OO) paradigm is intentional and aims to take advantage on some of the known characteristics of OO, such as modularity and aggregation based on roles.

We developed a proof-of-concept prototype as an evolution of the work for the quasi-manual approach, and in this section, we focus on the outcomes of the proposal, which were service guidelines, ASC and ESC. Unlike the service builder for the quasi-manual approach where policies and processes are targeted to XACML and BPEL, the ASC is based on generic description languages, not running technologies.

ASC is entirely written in OWL and particularly based on the Open Group SOA Ontology. However, since this ontology does not fulfill all the needs of the service code, it was extended. The ASC uses OWL-S for the specification of the process phase of the service refinement cycle because OWL-S contains language structures (e.g., decision and repetition) needed for representing a workflow.

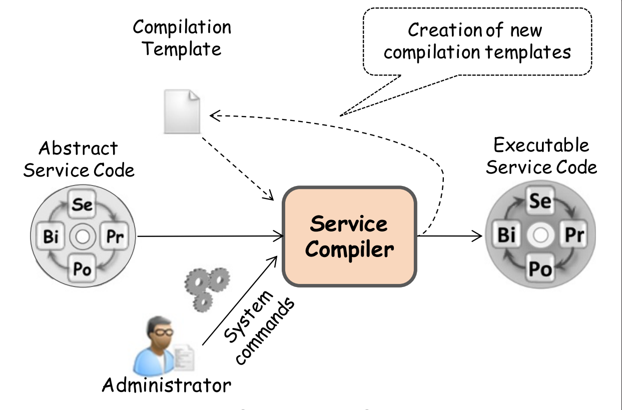


Figure 13 Service compiler inputs and outputs.

As far as the specification of abstract policies is concerned, there is an additional challenge involved in this task. Policy languages are usually tied to particular application domains, and even more general policy languages provide the means for binding their operation with environment. Therefore, the specification of abstract policies and their translation into executable policy language through the use of templates pose a new level of difficulty to the specification. Table 2 presents the main attributes for the specification of a generic policy considered for the ASC. An abstract policy language based on OWL and SWRL [23] was designed in order to have a generic description according to Table 2. SWRL has been chosen because it includes a high-level abstract syntax for rule definition in OWL, which gives more descriptive power to the language. Also, the specification of constraints is completely fulfilled by SWRL in our proposal. Since policies refer to the external world by calling services through the binding phase, all external references and details of particular application domains are performed by services.

As far as the specification of patterns is concerned, we think patterns should have the same structure as the outcomes whose creation they assist with. Once patterns can be considered 'incomplete' outcomes, the difference between them (i.e., between patterns and outcomes) is that patterns are identified by special tags in the syntax that must be properly replaced during the service creation process. For design patterns, these tags contain hints that must be replaced for the specification of a service guideline, whereas for implementation patterns, the tags represent blanks that must be filled according to the details of each service.

Specification of compilation templates is also an important part of a real deployment of a pattern-based service creation tool. In order for code generation to take place, the parsing for each particular executable code consists basically of two entities as defined in the template: boundaries and body. Boundaries are snippets that must be present at the beginning and at the end of the codes, according to the syntax of any particular

Table 2 Attributes of a generic policy description

Attribute	Description
Policy	Encapsulates all necessary information for the processing of rules
Target	Responsible for the matching (selection) during policy evaluation. Each policy has one target, and each target may contain many name-value attributes
Constraint	After the policy matching, more specialized rule verification must be done. A constraint may trigger many actions
Action	A description of an action to be taken
Attribute	A name-value entity responsible principally for the matching of policies

language. The body stands for the actual functionality as defined by the business analyst and implemented by the system developer. Each ASC fragment is translated into up to three types of executable code snippets: before location, in location, and after location. Before-location code is necessary for initializations, and it appears immediately after the boundary declaration. In-location code, as the name implies, refers to blocks placed exactly where they appear in relation to the other blocks, complying with the service specification in the ASC. After-location code is only used if the technology demands certain terminations after the work is done. Templates have pointers to assist with the positioning of those particular code fragments within the translated code. Figure 14 depicts the use of a compilation template as a translation map from ASC into ESC.

Other points worth highlighting about compiling templates and code generation concern the flexibility and compatibility of the code. As mentioned before, the templates are 'hardcoded' to the implementation patterns, meaning the structures of the template must be mapped to the programming language-specific structures. For this reason, it is expected that some executable languages may lack a compliant compiling template as well as that some programming structures be left unattended after template creation. About the compatibility of the compiled codes, this proposal advises the usage of a supporting platform which wraps the generated outputs in order to make them compatible among each other. In the case proposed in this paper, the service refinement cycle takes this role, mixing in a single execution environment the process workflows, policy entitling modules, and underlying systems such as databases.

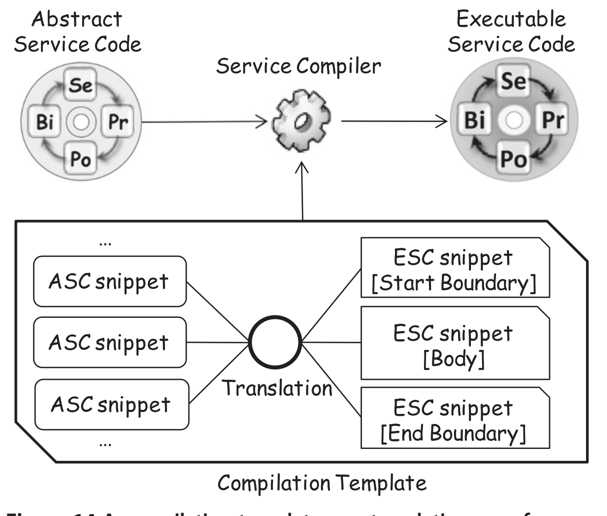


Figure 14 A compilation template as a translation map from ASC into ESC.

Case study: DVD delivery

Continuing the example in the 'Case study: DVD delivery' subsection of the 'Quasi-manual service creation' section, John is the business man which intends to offer a novel service. John should use a service designer in order to obtain a formalized description to be developed as the solution; nevertheless, differently from the quasi-manual approach, he is allowed to refer to design patterns, that is, he may follow an accepted model which is likely to assist at least a big part of his intentions, in the mentioned example, John should use a 'web delivery (DPT)' pattern.

Afterwards, John should address an experienced developer who is able to implement the solution in a complete yet generic manner, the abstract service code, so the solution is one step away from deployment. Aware that John used the mentioned pattern, Paul, the developer, may look for an implementation pattern which complies with John's choice and assists the implementation of the service components, through the service builder; in this case, he uses the 'DVD Delivery' guideline along with the 'web delivery (IPT)' pattern. The step in which the ASC is obtained is of paramount importance because of its relevance in this service description paradigm, and the paths from the ASC are two way: (1) it is a representative yet generic description of the service behavior, as designed by the entrepreneur and implemented by the developer; then, it can be shared as a formal description; and (2) it is ready to be compiled to any underlying technology, whereas there is an appropriate compiling template.

Hereinafter, the developer must take actions in order to make the ASC description executable and available. For that purpose, he must join a service provider company that offers possibly many different provisioning plans. Examples of these plans are WS-BPEL, Java or C, or others for workflow; XACML, Ponder, Rei, or others for policy entitling; and LDAP, SQL, XPeer, etc. for storage, among other standards that could be addressed in a more evolved service creation framework.

The advantage over quasi-manual creation approach is, additionally to the possible assistance by patterns, the division of deliverables in another format, the ASC, permits a bigger flexibility in deployment. Once the ASC is available, the entrepreneur may choose one among the many compiling templates, such as the 'BPEL_-XACML_SQL' template of the example, offered by the service provisioning company to get a customized service code tailored to be executed on service refinement cycle, which in turn will provide the abstraction layer that will merge together all the generated technology standards. Service compiler is the tool for this function, and the administrator from a provisioning company is the one responsible for this phase.

The whole process mentioned here can be found in Figure 15.

Similar to the previous scenario, this approach also tends to reduce development time and complexity due to the same aforementioned reasons. In addition, one should intuitively notice that it tends to improve the quality of the developed deliverables since the design and implementation patterns prevent incorrect service descriptions by suggesting the use of valid and proven solutions. In a first version of pattern-based approach, the observed gain in description provided by design and implementation patterns represented a step forward in the specification automation. A larger step is made possible by the insights coming from the generic abstract service code and the SiTra [30] approach, which leveraged the development of the fully automated part of the creation - the service compiler.

Results and discussion

Increasing levels of semi-automation

Semi-automated service creation represents a good balance between the manual and fully automated approaches mainly because a myriad of different techniques may be explored with increasing complexity between both extremes. Obviously, full automation is the 'Eldorado' that is sought after in service creation, yet it remains an open research problem. We advocate that experience with different levels of semi-automation will teach us important lessons that are still missing in the path towards full automation. Semi-automated service creation is feasible and may significantly contribute to accelerating the time to market for the creation of new services.

Different levels of automation within semi-automated techniques may require different skill sets and may provide different outcomes in terms of the user-friendliness of the tool, its power and flexibility in expressing business goals and technology characteristics, and its ability to create meaningful services. Lower levels of automation intrinsically demand more human interaction, whereas higher levels of automation reduce the need for user intervention, thus speeding up the service creation flow and providing cleaner outputs. Semi-automated techniques tend to generate a more robust code, for as long as the solutions prove to be correct, and they grant the user less power to interfere in the final lower level outcome. In effect, using a well-known design and implementation patterns, we expect the pattern-based approach to be less prone to errors.

For the proposed scenario, a fully automated approach, in our vision, would discard the expertise of the developer, being able to 'communicate' directly with the entrepreneur. That is to say, semantically described services would be able to compose with each other from high-level specifications. In other words, if the online available services (DVD rental and delivery) are properly described, with complete interfacing and behavior details, a tool - the service designer - would translate the semantics in service description to a human-friendly language. A possible user-tool interaction should be done through typed keywords. John defines the types, for example, in this scenario, 'DVD, rental, delivery, address, billing,' and semantic reasoner matches these keywords with service description found in a repository. A composition module, using the matching services, considers the inputs and outputs and shows to the user the possibly many compositions found, also in a high level way, so the user can finally select the workflow which is closest to the expected service (and then hire a developer to adapt the workflow, if necessary). A similar proposal to address automated service creation using ontology descriptions for the types was developed in [8].

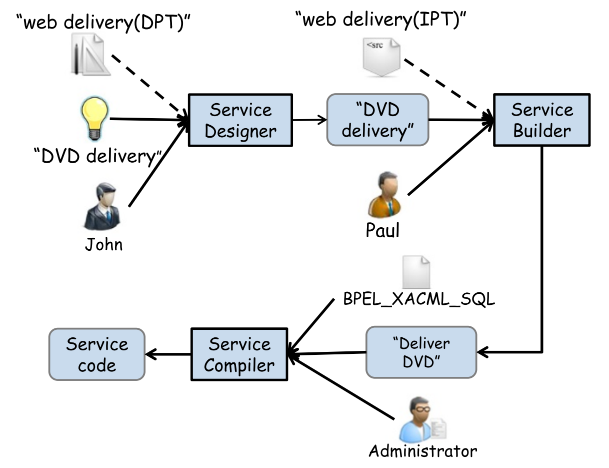


Figure 15 Use flow of pattern-based proposal to develop DVD delivery service.

Comparison: quasi-manual vs. pattern-based approach

We developed two approaches for service creation and found out that increasing the level of automation can cause a significant improvement for the activities related to service creation in terms of both human effort and expressiveness of the solution. The quasi-manual approach requires more manual effort to develop a new service, whereas the pattern-based one requires a more elaborate design and a more powerful tool in order to help service creators.

Both approaches require skilled users, though most likely with different skill sets, but the pattern-based approach tends to soften the burden for business analysts and system developers by requiring less human intervention. The quasi-manual approach requires the business analyst to have a clear idea of the new service, including the business goals and processes involved, when operating the service designer tool. The system developer needs a simple and easy-to-understand service guideline in order to 'program' the service. On the other hand, within the pattern-based approach, the design patterns may help the business analyst with the business process, thus freeing him to focus on the service goals and making the job more straightforward for the system developer. As far as the skills needed for using the service builder are concerned, both approaches may require experienced system developers who can map a service guideline into an abstract or executable service code.

All minimally automated service creation approaches will require some effort for setting up the environment. For the service builder, the most noticeable support feature is a service directory with semantic discovery and matching capabilities. This is a challenging problem and therefore is considered to be out of the scope of this paper, as we assumed in our prototype that all services needed by the system developer might be easily found. The pattern-based approach also requires libraries of existing design patterns, implementation patterns, and compilation templates. Otherwise, a real deployment will demand much effort during the initial phase because patterns and templates will have to be created from scratch.

We expect that by increasing the levels of automation, the service creation life cycle will be more efficient and agile, thus decreasing the time to market for new services. In this regard, the quasi-manual approach heavily depends on the skills of its users, which makes it more susceptible to errors that will require corrective maintenance. On the other hand, multiple development cycles may be avoided by the pattern-based approach, as long as its users are able to correctly use design and implementation patterns, which are expected to be less error prone. Also, service compilation is a completely automated phase that barely influences the whole service creation process. With these slow steps of improvement, one divides the service creation task in smaller problems in which it is simpler to perceive how automation can be increasingly merged to the overall creation activity. This insight inspires moving on in solving the smaller problems priorly than attempting to embrace the whole creation once the results are more straightforward and prone to be understood and applied.

Service specification formats

The existence of a well-defined service specification format, the service code, plays a paramount role in the design of service creation approaches and tools. The well-known and well-accepted service code makes the expected outcomes of each phase - and the software components used in the tools - clear.

Service creation should be performed without a need to bind the solution to specific technologies at first, like in our pattern-based approach, where an intermediate abstract service code is generated. A service specification should contain the main components that consistently define its behavior, constraints, and interfaces and should be independent of the environment it is going to be executed on. This approach can foster the interchange of services - which can be run in different data centers in the cloud - between different organizations. The translation of the abstract service code into executable languages and standards should be left to the very last moment, immediately before deployment.

Because execution engines and specification languages do not follow a single standard, we think that an approach that adds flexibility to service development and deployment is welcome. That is to say, a unified, generic description of the outputs (using ontologies, for instance) plays such an important role because it permits the sharing of a business idea and its deployment to many possible technologies.

Next steps in semi-automation

Based on our experience, we aim at advancing the levels of automation for semi-automated service creation techniques and tools. Some features may be added to the pattern-based approach to increase its levels of automation and make easier the job of service creators:

- Strengthening the connecting between design and implementation patterns. For example, the tool may automatically suggest one (or various) implementation pattern given an existing design pattern.
- Providing auto-completion features for automatically suggesting services that may be used in a certain process.
- Adding reasoning features to particular parts of the service builder that will automatically suggest services that fit in the process. Those features should be added carefully on a step-by-step basis in order to make it useful for service creators.
- Creating patterns based on the analysis of common existing service compositions. The service creation tool might suggest new patterns based on a network theory analysis of services, similar to the analysis of scientific workflows in [31].

Critical evaluation

Despite the advances advocated by this proposal and the results illustrated in this work, some downsides of the approach must be presented as well. The most relevant limitation takes place because of the most advanced feature of the proposal - the compiling templates are not able to translate to every programming language. Once the proposal is targeted at fostering compositions, final service codes (i.e., executable service code) whose process programming language is dissimilar or incompatible with the workflow paradigm should not be expected as possible outputs in this proposal.

The approaches described in this paper were focused on service composition, i.e., allowing the designer to coordinately connect available subservices in order to construct a service with more final value; therefore, the proposal should preferably be used to this end. Services with extreme performance requirements, such as those that are process intensive or depending on legacy resources, are not observed. Since third party services are usually being part of the composition, quality of service for the final service is also an unattended issue.

The motivation that led us to have a web-based service creation proposal limited the interaction ability of the developer. The tool interacts at a high level, and if advanced details of service components' description are necessary, they must be done in the executable service code level in the appropriate development environments. This depletes all the advocated advantages of the abstract service code. In order to solve this issue, an advanced manner to interact with the generated ASC has to be done and remain fully compliant with the compiling templates and the service compiler.

Conclusions

In this paper, we highlight the idea that continuously adding automated features to a service creation tool may contribute significantly to understanding the main problems and challenges of this area. Consequently, this approach will help to build up the knowledge needed to enable a move towards higher levels of automation by transferring parts of the human effort to the existing tools.

We proposed two approaches for service creation and developed tools for them. We found out that increasing the level of automation can bring about a significant improvement for the activities related to this activity, in terms of both human effort required and the expressiveness of the solution. The quasi-manual approach requires more manual effort to develop a new service, whereas the pattern-based one requires a more elaborate design and a more powerful tool in order to help service creators. Both are based on the service refinement cycle, generate services in the form of service code, and are based on a generic platform of a service creation tool, which makes the job of executing and comparing different approaches easier.

In terms of future work, we intend to continue developing new approaches that add new automated features so that we can compare more than two approaches. Also, we will work towards defining metrics for and performing qualitative and quantitative comparisons among different approaches for service creation.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

EA carried out this work along with his master degree thesis, performing state-of-the-art studies, implementing, testing and reporting the presented solutions; CK conceived the study, participated in its design and coordination, helped to draft the manuscr ipt and helped to revise its final version; RD collaborated with the state-of-the-art research, gathering background technologies and assisted the development of the prototypes; BO participated in meetings to discuss and define the methods, contributing with ideas for requirements of the solutions; DS participated in the design of the study, discussed about alternative solutions which aided the definition of the architecture of the implementations. All authors read and approve the final manuscript.

Authors' information

EA received his master's degree in Computer Science from the Federal University of Pernambuco (Recife, PE, Brazil) in 2010. He is currently an assistant research fellow in the Networking and Telecommunication Research Group (GPRT) at the Federal University of Pernambuco. CK received his Ph.D. in Computer Science from the Federal University of Pernambuco (Recife PE, Brazil) in 2003. He is an associate professor of computer networks at the Federal University of the ABC in Santo André SP, Brazil where he currently holds the position of provost for graduate studies. His research interests include service-oriented computing, cloud computing, service composition, policy-based management, Internet traffic analysis, and complex networks. He is also a senior research fellow in the Networking and Telecommunication Research Group (GPRT) at the Federal University of Pernambuco. RD received his Ph.D. in Computer Science from the Federal University of Pernambuco (Recife PE, Brazil) in 2012. He currently holds a professorial position at the Federal Institute of Paraíba, in Campina Grande, Brazil and is an assistant research fellow in the Networking and Telecommunication Research Group (GPRT) at the Federal University of Pernambuco. His research interests include service-oriented computing, service composition, autonomic computing, and network management. BO has a Master of Science degree in Computer Science and History of Ideas and Sciences from Uppsala University, Sweden. He started in telecommunications when he joined Ericsson in the late 80s. In the mid 90s, he moved to IP technology, especially QoS, and was active in IETF in the standardization of DiffServ and policy-based networking. Since 2005, he has focused on future networking technologies. DS received his Ph.D. degree from Kent University in 1990. From 1990 to 1992, he was a research fellow in the Computer Science Department, University College London. He is currently a professor at the Computer Science Department of the Federal University of Pernambuco, Brazil. He is one of the cofounders of GPRT, a research group in the areas of computer networks and telecommunications. His current research interests include traffic engineering of IP networks, wireless communications, broadband access, and network management. He is a senior member of the IEEE Communications Society and currently leads a number of research projects.

Acknowledgements

This article and the thesis where this originated from were supported by the Network and Telecommunications Research Group, from the Informatics Centre in Federal University of Pernambuco, Brazil. Its funding was provided by Ericsson Research, Sweden, which kindly endorsed the research inside one of its ongoing projects and authorized its publication as well.

Author details

[1]Federal University of Pernambuco, Professor Moraes Rêgo Av., 1235, Cidade Universitária, Recife 50670-901, Brazil. [2]Federal University of ABC, Santa Adélia St., 166, Bangu, Santo André 09210-270, Brazil. [3]Ericsson Research, Torshamnsgatan, 23, Kista, Stockholm 164 83, Sweden.

Received: 26 September 2013 Accepted: 7 November 2013
Published: 23 January 2014

References

1. Kamienski CA, Dantas R, Fildago J, Sadok D, Ohlman B (2010) Service creation and execution with the service refinement cycle. In: NOMS 2010. Network Operations and Management Symposium IEEE, Osaka, April 2010. IEEE, Piscataway, pp 829–832, 63

2. Erl T (2007) SOA Principles of service design, 1st edn. Prentice Hall, Boston
3. Estefan JA, Laskey K, McGabe FG, Thornton D (2008) Reference Architecture for Service Oriented Architecture – Version 1.0. OASIS, 23 April 2008., Available at: http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra-pr-01.pdf. Accessed 8 April 2009
4. Curbera F, Duftler M, Khalaf R, Nagy W, Mukhi N, Weerawarana S (2002) Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI. IEEE Internet Comput 6(2):86–93
5. Peltz C (2003) Web services orchestration and choreography. Comput Mag IEEE 36(10):46–52
6. Bandara AK, Lupu EC, Moffett J, Russo A (2004) A goal based approach to policy refinement. In: POLICY 2004. Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, New York, 7–9 June 2004. IEEE, Piscataway, pp 229–239
7. Rao J, Su X (2004) A survey of automated web service composition methods. In: Cardoso J, Sheth A (eds) Semantic web services and web process composition. First International Workshop, SWSWPC 2004, San Diego, July 2004. Lecture notes in computer science, vol 3387. Springer, Heidelberg, pp 43–54
8. Dantas R, Azevedo E, Dias C, Lima T, Sadok D, Kamienski CA, Ohlman B (2011) Facilitating service creation via partial specification and automated composition. IEEE World Congress on Services Composition Workshop, Washington, DC, July 2011. IEEE, Piscataway, pp 303–310
9. Gannod G, Mudiam S, Lindquist T (2004) Automated support for service-based software development and integration. J Syst Software 74:65–71
10. Agarwal V, Chafle G, Mittal S, Srivastava B (2008) Understanding approaches for web service composition and execution. In: COMPUTE '08. Proceedings of the 1st Bangalore Annual Compute Conference, Bangalore, January 2008. ACM, New York
11. Jin L, Pan P, Ying C, Liu J, Tian Q (2009) Rapid service creation environment for service delivery platform based on service templates. In: IM '09. 11th IFIP/IEEE International Symposium on Integrated Network Management, Long Island, June 2009. IEEE, Piscataway, pp 117–120
12. Syu Y, Ma SP, Kuo JY, FanJiang YY (2012) A survey on automated service composition methods and related techniques. 2012 IEEE Ninth International Conference on Services Computing (SCC), Honolulu, June 2012. IEEE, Piscataway, pp 290–297
13. Paulson LD (2003) Building rich web applications with Ajax. IEEE Computer 38(10):14–17
14. GWT Project (2006) Google, Inc., Mountain View., http://code.google.com/webtoolkit. Accessed 8 May 2013
15. The Eclipse Foundation (2013) Eclipse Foundation, Inc., Ottawa., http://www.eclipse.org. Accessed 8 May 2013
16. Hayes B (2008) Cloud computing. Commun ACM 51(7):9–11, 71
17. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson DA, Rabkin A, Stoica I, Zaharia M (2009) Above the clouds: a Berkeley view of cloud computing. Technical report (UCB/EECS-2009-28), University of California, Berkeley
18. The Open Group (2010) Service-oriented architecture ontology, technical standard, draft 3.2., http://www.opengroup.org/soa/source-book/ontology. Accessed 7 May2013
19. McGuinness DL, van Harmelen F (2009) OWL 2 web ontology language document overview. W3C recommendation., http://www.w3.org/TR/owl2-overview. Accessed 22 Jan 2012
20. OASIS (2007) Web services business process execution language, version 2.0. OASIS, Burlington
21. Martin D, Burstein M, Hobbs J, Lassila O, McDermott D, McIlraith S, Narayanan S, Paolucci M, Parsia B, Payne T, Sirin E, Srinivasan N, Sycara K (2004) OWL-S: semantic markup for web services. W3C, Cambridge, MA, USA
22. Parducci B, Lockhart H, Levinson R, McRae M (2005) Extensible access control markup language, version 2.0. OASIS standard., http://www.oasis-open.org/committees/xacml. Accessed 22 Jan 2012
23. W3C (2004) SWRL: a semantic web rule language combining OWL and RuleML. SWRL project homepage., Available at http://www.w3.org/Submission/SWRL Accessed 8 May 2013
24. Kamienski CA, Fidalgo J, Dantas R, Sadok D, Ohlman B (2007) XACML-based composition policies for ambient networks. In: POLICY '07. Eighth IEEE International Workshop on Policies for Distributed Systems and Networks, Bologna, June 2007. IEEE, Piscataway, pp 77–86
25. MacKenzie CM et al (2006) Reference Model for Service Oriented Architecture – Version 1.0, OASIS, 2006., http://www.oasis-open.org/committees/soa-rm, accessed 27/09/2011
26. Workflow Management Coalition (2008) XML process definition language (XPDL), WFMC-TC-1025, version 2.1. Workflow Management Coalition, Hingham
27. Damianou N, Dulay N, Lupu E, Sloman M (2001) The ponder policy specification language. In: Sloman M, Lupu E, Lobo J (eds) Policy for distributed systems and networks. International Workshop, POLICY 2001 Bristol, UK, January 29–31, 2001. Lecture notes in computer science, vol 1995. Springer, London, pp 18–38
28. Kagal L, Finin T, Joshi A (2003) A policy language for a pervasive computing environment. In: POLICY '03. Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks, June 2003. IEEE, Washington, DC, p 63
29. Erl T (2009) SOA Design patterns, 1st edn. Prentice-Hall, Boston
30. Bordbar B, Howells G, Evans M, Staiopoulos A (2007) Model transformation from OWL-S to BPEL via SiTra. In: ECMDA-FA'07. Proceedings of the 3rd European Conference on Model Driven Architecture-Foundations and Applications, Haifa, June 2007. Lecture notes in computer science, vol 4530. Springer, Heidelberg, pp 43–58
31. Tan W, Zhang J, Foster I (2010) Network analysis of scientific workflows: a gateway to reuse. IEEE Comput 43(9):54–61

doi:10.1186/1678-4804-20-2
Cite this article as: Azevêdo *et al.*: **A path to automated service creation via semi-automation levels.** *Journal of the Brazilian Computer Society* 2014 **20**:2.