Validation of Service Blueprint Models by Means of Formal Simulation Techniques

Montserrat Estañol[2], Esperanza Marcos[1], Xavier Oriol[2], Francisco J. Pérez[1], Ernest Teniente[2], and Juan M. Vara[1(✉)]

[1] Kybele Research Group, University Rey Juan Carlos, Madrid, Spain
{esperanza.marcos,francisco.perez,juanmanuel.vara}@urjc.es
[2] Universitat Politàcnica de Catalunya, Barcelona, Spain
{estanyol,oriol,teniente}@essi.upc.edu

Abstract. As service design has gained interest in the last years, so has gained one of its primary tools: the Service Blueprint. In essence, a service blueprint is a graphical tool for the design of business models, specifically for the design of business service operations. Despite its level of adoption, tool support for service design tasks is still on its early days and available tools for service blueprint modeling are mainly focused on enhancing usability and enabling collaborative edition, disregarding the formal aspects of modeling. In this paper we present a way to support the validation of service blueprint models by simulation. This approach is based on annotating the models with formal semantics, so that each task can be translated into formal logics, and from them, to executable SQL statements. This works opens a new direction in the way to bridge formal techniques and creative service design processes.

Keywords: Service blueprint · Validation · Simulation

1 Introduction

Beyond the computational point of view, services have been a matter of interest for the academia since the appearance of the first studies on services marketing in the early 50's [1] to the advent of Service Science from IBM [2]. More recently, the fact that approximately 60% of the world's workforce is currently employed by either public or private branches of the Service Sector and this value rises to 80% in developed countries has significantly contributed to renew the interest in services and related disciplines.

One of those disciplines is Service Design, which aims at helping in the development or improvement of services in order to deliver user-centered services by focusing on the interactions (or touchpoints) between the provider and the consumer [3]. Its main principles are: human-orientation, value co-creation, process-based nature, tangible evidences and holistic view. Born also in the context of research on services marketing, service design evolved and gained impact through

M. Maximilien et al. (Eds.): ICSOC 2017, LNCS 10601, pp. 80–95, 2017.
https://doi.org/10.1007/978-3-319-69035-3_6

the impulse of *IDEO*[1] and has finally been established as the entry point to service development for any organization seriously concerned about user experience, digital transformation and the like (see for instance the efforts on this issue of the British government around the *Government Digital Service*[2]).

The most popular service design technique is service blueprinting [4]. In essence, a (service) blueprint is a graphical tool to visualize the different parts of a given service and the interactions between the stakeholders of such service. In contrast with BPMN, an organization-focused notation for business process modeling, service blueprinting is a user-centered approach to business process modeling. This has turned to be key for service designers in the digital age, where most of the innovation has to happen at the touch points between provider and consumer.

Despite the fact that service blueprinting was originally intended to enable a more rigorous control and analysis of service delivery, it has partially failed since despite its widespread adoption, it is most commonly used as an sketching tool to provide first-draft solutions but they are rarely used to support any kind of formal reasoning.

By contrast, bringing some degree of formability to service blueprinting has proven to contribute to ensure the success of the process and in turn increases the effectiveness of the blueprint, improving the rationality of the decisions within the company [5]. As a matter of fact, even though there exists a number of proposals to bring formalization to other existing techniques for business process modeling exist (see [6,7] for instance), to the best of our knowledge there exists no similar proposal for service blueprinting.

The main goal of this paper is to introduce a framework that permits formally defining service blueprints, and validating them. To do so, we propose to, during the definition of the service blueprint, annotate its tasks with some formal semantics. Thus, each task unambiguously specify its behavior. As a result, we can validate the service blueprint by interpreting such semantics, simulating its execution, and checking that no undesirable situation occurs during the simulation.

In order to support the full process of this framework, we extend the *INNoVaServ*[3] modeling tool, and propose its integration with the *OpExec* simulating library[4]. The former is an EMF-based toolkit [8] with a visual DSL for service blueprint modeling, thus, offering very good capabilities for easily defining service blueprints. The latter is a Java library that permits simulating processes described in formal logics, while checking validation conditions (aka integrity constraints). Thus, the integration of both tools covers our framework entirely, permitting the definition and formal validation of service blueprints.

[1] https://www.ideo.com/.
[2] https://gds.blog.gov.uk/.
[3] http://www.kybele.etsii.urjc.es/innovaserv/index.php.
[4] http://www.essi.upc.edu/~xoriol/opexec/.

2 Context

This section presents the research context of this work. To that end, we first summarize service blueprinting notions, to later introduce *INNoVaServ*, the toolkit in which to integrate the process simulation capabilities.

2.1 Service Blueprinting

The service blueprint is a graphical tool for the design of business models, specifically for the design of business service operations, which is focused on detailing the interaction between the customer and the service provider in the provision of a given service [9]. Being a tool for service design and giving the process nature of services, service blueprinting is actually another technique for process modeling. The notably difference being in this case the focus on the customer experience, which is clearly illustrated by making explicit the touchpoints, the *physical* evidences related with the provision of the service and the limits between frontstage and backstage.

For instance, Fig. 1 shows an excerpt[5] of the renting process of *car2go* service blueprint: the user needs to rent a car and therefore he turns to the *car2go* app (online/physical evidence); a couple of consumer-provider interactions take then place along the *line of interaction*. Data provided by the user is then checked in the *backstage* whereas external entities are contacted to order vehicle maintenance and process payments.

As can be shown, a service blueprint is composed by five lanes or regions of activity that help to distinguish those actions that are specific to the service provider from those performed by the customer/consumer. Such lanes are listed below from top to bottom:

- *Physical Evidence.* This region represents the evidences, facts or global actions that give rise to the interaction between the customer and the service provider. A user who has the need of renting a car for private transportation to go somewhere is something that gives rise to the interaction between the customer and *car2go*.
- *Attendee Action.* It is devoted to describing the actions that a client performs while interacting with the *front-end* of the service, like selecting the car to rent or notifying *car2go* when he has arrived to the destination. Lower bound of this region is called *Line of Interaction* and it detaches the actions performed by the customer from those performed by the service provider.
- *Frontstage Interactions.* The activities performed by the service provider which entail some type of interaction with the customer are represented here. Asking the user to select a car or to check his *car2go* account are examples of this kind of activities.

[5] Full version can be found at http://www.kybele.es/publications/-car2go_rentingProcess_extended.png.

– *Back of Stage Interactions.* The activities performed in the shadow by the provider to operate the service, like updating the vehicle status when a car is rented, are represented in this region. These are actions needed to deliver the service but which the customer cannot see or interact with.
– *Support Processes.* Actions supporting the service, sometimes performed by third parties, are represented here. The interaction between the maintenance company and *car2go* to order vehicles maintenance is one of those actions that remain hidden for the customer.

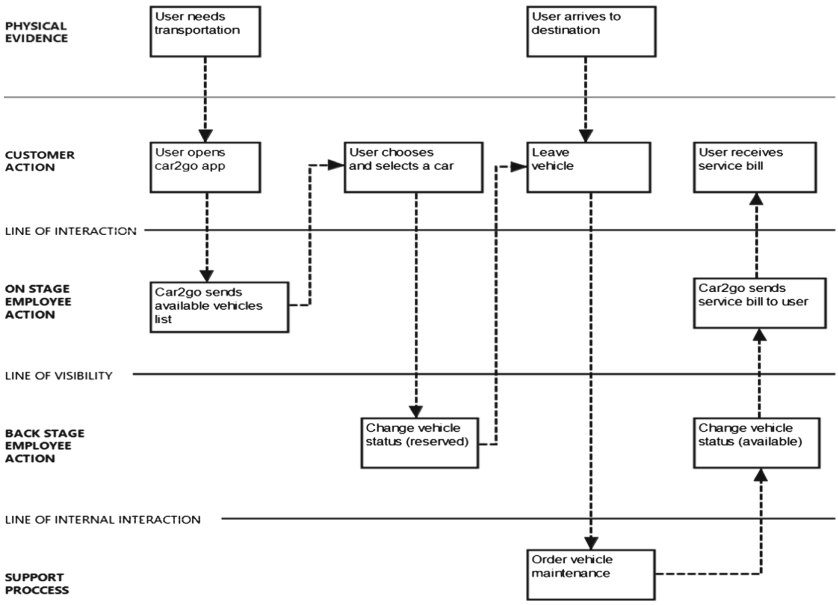

Fig. 1. Excerpt of a service blueprint made with INNoVaServ - *car2Go* renting process

As the next section will show, service blueprint diagrams can be faithfully represented with the modeling environment provided by *INNoVaServ*.

2.2 Introducing INNoVaServ

INNoVaServ[6] is a modeling environment for the design of business models and service process operations which, to date, supports 4 different notations: Canvas [10], e3Value [11], Process Chain Network (PCN) [12] and the Service Blueprint.

It is a first step towards solving the lack of proper tool support for bridging existing business modeling notations. To that end, *INNoVaServ* integrates different tools to register and manage the relationships between models defined with

[6] http://www.kybele.etsii.urjc.es/innovaserv.

the different techniques for business (process) modeling. In the medium term, the aim is at automating the identification of such relationships and enlarge the number of notations supported.

Basically, the toolkit can be thought of as a set of four integrated visual DSLs, one for each modeling notation supported by the tool. Each of such DSLs was developed atop EMF and GMF [8] following the guidelines sketched in [13] for the development of model-based tools that take the shape of DSL toolkits.

It is worth noting that due to the fact that it has been entirely developed atop of EMF/GMF, it is immediately interoperable with any other EMF/GMF based tool. Since EMF/GMF has converted in the de-facto standard for the development of model-based tooling, the scope of tools with which *INNoVaServ* can then interoperate is huge. The different tools supporting BPMN, like the Eclipse BPMN Modeler[7] or the Obeo BPMN Designer[8] (there is indeed many others BPMN editors based on EMF/GMF), Papyrus UML and any other EMG/GMF-based editor, etc.

3 Enabling Formal Verification of Service Blueprint Models in *INNoVaServ*

This section describes our approach for enabling formal verification of service blueprint models. We start by showing the architecture for integrating INNoVaServ with the OpExec tool, which will be the basis for this verification. Then, we explain our proposal for formally defining service blueprint tasks and we show how to achieve the intended formal semantics for validation.

3.1 Functional Architecture and Design

In Fig. 2 we summarize the architecture of *INNoVaServ* (according to the convention widely adopted by Eclipse developers to represent the architecture of their proposals), together with our proposed integration with the OpExec simulation tool.

The technological basis of *INNoVaServ* is Eclipse. DSLs are mainly developed atop of EMF/GMF while some minor refinements coded with the aid of JFace and SWT to obtain the desired functionality from diagrammers. *INNoVaServ* can validate its defined processes by means of bringing them to the OpExec tool, and simulating the execution of its tasks. In its turn, OpExec works by storing in memory a logic representation of the process to simulate, and persisting the data of the process in a relational database.

3.2 Formally Defining Service Blueprint Tasks

Service blueprints, as they are, provide an intuitive and understandable overview of the different tasks and activities required to provide a service or achieve a

[7] http://www.eclipse.org/bpmn2-modeler/.
[8] http://marketplace.obeonetwork.com/module/bpmn.

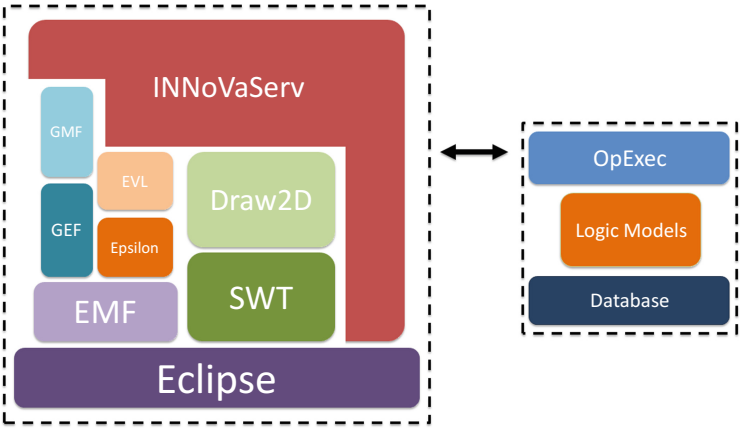

Fig. 2. Overview of technological dependencies of INNoVaServ and OpExec

certain goal. However, there is no formal meaning attached to each of the tasks. From their names, we may have an intuitive idea of what they imply, but we do not know exactly what it is they do. Therefore, if we wish to validate the model considering what the tasks do, it is necessary to enrich the initial service blueprint as explained below.

In order to provide the tasks with meaning, it is necessary to have an underlying data model, to represent the relevant information that will be manipulated by them. We propose using a UML class diagram for this purpose. Figure 3 shows the class diagram for the service blueprint in Fig. 1.

This class diagram keeps information about the company's *Vehicles* (id, status, battery level, etc.) together with their *Location* and the *ServiceBills* which have resulted from the use of the *Vehicle*. The system also stores information about which *User* is currently using or has booked a *Vehicle* (if any) and the *ServiceBills* of a certain *User*. Apart from the *User* basic information, such as id, name or accountStatus, the system also keeps track of the *BankAccounts* of *User* and their funds.

The data represented in the class diagram should satisfy some additional conditions (aka integrity constraints), in order to ensure the correct behavior of the service. For instance, each UML class could have an identifier (i.e. "primary key"). In our example, *Vehicle*, *User*, *ServiceBill* and *BankAccount* are identified by their *id*. *Location*, on the other hand, is identified by its *coordinates*. More complex conditions might be observed. For example, *damaged Vehicles cannot be booked by a User*.

Then, given the data model, it is possible to specify unambiguously what each of the tasks is doing. Our approach will use structured natural language for this purpose.

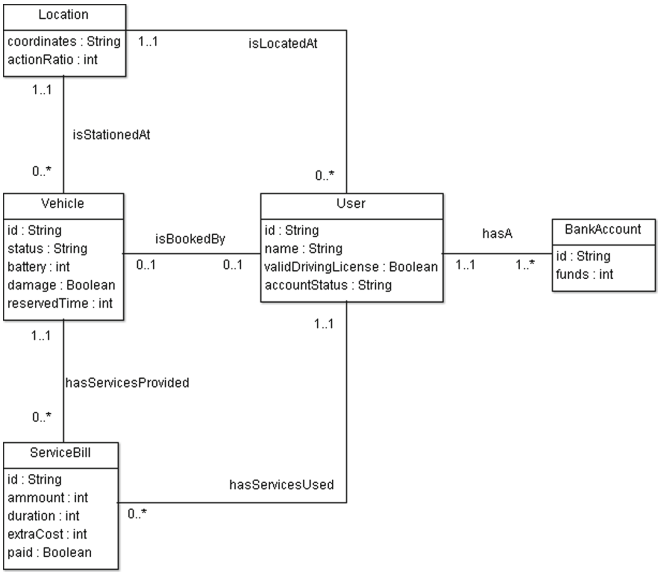

Fig. 3. Class diagram representing the underlying data model for *car2go*.

Structured Natural Language. In order to formalize the meaning of the tasks in the service blueprint we propose using structured natural language. This language is based on using a few keywords and following certain patterns, which result in sentences which can be easily understood. By using these it is possible to state selection and basic changes over data (creation, updates, deletions). Hence, we strike a balance between understandability, simplicity and expressibility.

The grammar defining this structured language is the following:

select *ClassName* [**with** Attributes]
delete_c *ClassName*
ClassAction *ClassName* Attributes

AssociationAction *AssociationName* (*ClassName*, Classes)

ClassAction → **check** | **change** | **create**
AssociationAction → **associate** | **delete_a**

Attributes → *attributeName(value)* | *attributeName(value)*, Attributes
Classes → *ClassName* | *ClassName*, Classes

Keywords are in bold. Class and attribute names are in italics, and should be replaced by any class or attribute name in the model, respectively. Square brackets represent optional parameters. *value* should be replaced by either a value (e.g. string, integer) or an input parameter, the latter representing an input value provided by the user.

select and **check** both refer to conditions that must be true of the particular object they are applied to. **check** refers to an object that has been obtained previously and which must fulfill a certain condition. **select**, on the other hand, obtains a new object which had not been obtained previously; **select**...**with**... obtains a new object which fulfills the conditions stated in the **with** (which will refer to its attributes).

The remaining keywords correspond to changes made to the underlying data. **create** will create a new instance of a class with the given attribute values. **delete_c** will delete the given instance of a class. **change** will update an attribute (or several) of a class to the given values or input parameters.

Similarly, **associate** will create an instance of the named association with the indicated classes. **delete_a** will delete the instance of the association with the given name and the participating classes.

This results in statements such as the following, where the first corresponds to task *User chooses and selects a car* and the second to *Change Vehicle Status (reserved)*:

select *Vehicle* **with** *status('available')*
change *Vehicle status('reserved')*

We assume that references to a class point to an instance or object of the class in question which has obtained previously. Therefore, in the previous example statements refer to the same vehicle.

3.3 Executing the Formal Semantics for Validation

Once the service blueprint is annotated with the formal semantics, it is unambiguous enough to validate its execution. In particular, we aim at ensuring that, when executing the process, its data state never becomes inconsistent (i.e., never stores a state that cannot occur in the real world). For doing so, we need to define some *integrity constraints*, that is, some conditions that *consistent data states* always satisfy, thus, any violation of a constraint points an inconsistency.

Hence, our validation approach consists in (1) translating such annotations into an executable language, (2) run the process, and (3) check that such execution satisfies our defined set of integrity constraints.

For our purposes, we use as executable language (a subset of) the executable logic rules stated in [14]. Such rules can be executed by means of a prototype tool we call OpExec, which essentially persists the data of the process into a relational database, and checks that such data satisfy a set of user-defined integrity constraints. In this manner, if the OpExec tool detects a violation of some of these constraints, we can realise that the service is ill-defined.

In the following, we first present the executable logic rules we use. Then, we show how to translate the previous patterns annotated in the service blueprint into such logics. Finally, we show how to validate the service blueprint by running these rules over the OpExec tool.

Executable Logic Rules. In our particular case, the executable logic rules are some rules following one of these forms:

$$ins\_C(\overline{x}) : -taskName(), arg_0(x_0), ..., arg_n(x_n)$$
$$ins\_Select\_C(x) : -taskName(), arg_0(x_0), ..., arg_n(x_n), C(\overline{x})$$
$$del\_C(\overline{x}) : -taskName(), Select\_C(x)$$
$$del\_Select\_C(x) : -taskName(), Select\_C(x)$$
$$ins\_R(\overline{x}) : -taskName(), Select\_C_0(x), ..., Select\_C_n(x)$$
$$del\_R(\overline{x}) : -taskName(), Select\_C_0(x), ..., Select\_C_n(x)$$
$$query(\overline{x}) : -taskName(), arg_0(x_0), ..., arg_n(x_n), C(\overline{x})$$

Intuitively, the first two rules state that an insertion/selection of an instance of class C should be realized if the task called *taskName* is invoked with arguments $x_0, ..., x_n$, where such arguments are used to specify the values of attributes of the object being created/selected. In addition, the second rule forces the instance of C to exists in order to be selected. Similarly, the third and forth rule states a deletion/deselection of an instance of C that was previously selected. Then, the fifth and sixth rules state that a creation/deletion of an association R with the selected objects should be performed. The last rule is only a query to check the existence/inexistence of some instance of C.

Translating Natural Language Patterns into Executable Logic Rules. Now, we show how to translate the natural language patterns used to annotate the service blueprint into such executable logic rules.

Intuitively, each natural pattern presented is mapped to one or more executable rules. This is because, for instance, the creation/deletion of an object in some class C might encompass the creation/deletion of the same object in its sub/superclasses C'/C'', and each creation requires its own executable logic rule.

Table 1 summarizes these mappings. For each task in the service blueprint with its corresponding annotation in natural language, we generate the executable logic rules stated in the right column. In particular, the task name of the pattern brings the name to the *taskName()* atom of the rule, and each user given value v_i in the pattern originates a $arg_i(v_i)$ atom. As expected, the atoms using classes/associations C/R take its name from the classes/associations C/R used in the pattern, and user-defined constants from the pattern are propagated to the rule.

For instance, in our example, the annotation of the tasks *User chooses vehicle* and *Change vehicle status* would be translated into:

$ins\_Select\_Vehicle(v)$:- $UserChoosesVehicle()$, $Vehicle(v, \text{'available'}, b, d, rt)$
$del\_Select\_Vehicle(v)$:- $UserChoosesVehicle()$, $Select\_Vehicle(v)$
$ins\_Vehicle(v, \text{'reserved'}, b, d, rt)$:-$ChangeVehicleStatus()$, $Select\_Vehicle(v)$, $Vehicle(v, s, b, d, rt)$
$del\_Vehicle(v, s, b, d, rt)$:- $ChangeVehicleStatus()$, $Select\_Vehicle(v)$, $Vehicle(v, s, b, d, rt)$

Table 1. Natural language patterns to executable logic rules

N.L. Pattern	Derivation rules to create
create C	ins_C(c, v_0, ..., v_n) :- taskName(), $\arg_0(v_0)$, ..., $\arg_n(v_n)$
$at_0(v_0)$, ..., $at_n(v_n)$	ins_C'(c, v_0, ..., v_0) :- taskName(), $\arg_0(v_0)$, ..., $\arg_n(v_n)$; for each C $\sqsubseteq$ C$'$
delete_c C	del_C(c) :- taskName(), Select_C(c)
	del_C'(c) :- taskName(), Select_C(c), C$'$(c); for each C$'$ $\sqsubseteq$ C
	del_C''(c) :- taskName(), Select_C(c); for each C $\sqsubseteq$ C$''$
associate R(C_0,...,C_n)	ins_R(c_0,...,c_n) :- taskName(), Select_C_0(c_0), ..., Select_C_n(c_n)
delete_a R(C_0,...,C_n)	del_R(c_0,...,c_n) :- taskName(), Select_C_0(c_0), ..., Select_C_n(c_n)
change C at(v_i)	ins_C(c, ..., v_i, ...) :- taskName(), $\arg_0(v_i)$, Select_C(c), C(c, v_0, ..., v_n)
	del_C(c) :- taskName(), Select_C(c)
select C with	ins_Select_C(c) :- taskName(), $\arg_0(v_0)$, ..., $\arg_0(v_n)$, C(c, v_0, ..., v_n)
$at_0(v_0)$, ..., $at_n(v_n)$	del_Select_C(c) :- taskName(), Select_C(c)
check C	query(v_0, ..., v_n) :- taskName(), $\arg_0(v_0)$, ..., $\arg_n(v_n)$, C(c, v_0, ..., v_n)
$at_0(v_0)$, ..., $at_n(v_n)$	

The first rule selects a vehicle that is available when the user executes the task *UserChoosesVehicle*. The second rule is used to deselect any other previously selected vehicle. The third and forth rule are in charge of updating the state status of the selected vehicle to "reserved" when the user executes the *ChangeVehicleStatus* task.

Validation Through Executing the Logic Rules. The idea now is to use the OpExec tool to (1) load the executable logic rules representing the business process tasks, (2) load some integrity constraints to check while executing the process, and (3) execute the process to validate the satisfaction of the constraints.

In order to load the executable logic rules, OpExec only needs the rules themselves, and some relational database connection containing one table for each class/association, and one *Select_C* table for each class C in order to store the current instances selected for each class.

OpExec can then load integrity constraints written in the of form *denial constraints*, that is, logic formulas stating the condition that should never occur in the database. For instance, the condition *damaged Vehicles cannot be booked by a User* can be written as

$$\bot \text{ :- } Vehicle(v, s, b, d, rt), s = \text{'reserved'}, d = \text{'true'}$$

Then, at runtime, OpExec is in charge of executing the logic rules according to the client invocations (*INNoVaServ*, in this case). Such invocations cause the insertion/deletion of objects in the database, or their selection (which is stored in the corresponding *Select_C* table), according to the translation of the natural language patterns. The *check* pattern requires special attention since it is translated as a new query that checks the condition. In this case, OpExec executes the query and returns the result to the client, so, the client can take the decision of what to do next (such as repeating the last task if the checking did not succeed).

The important feature of OpExec w.r.t. validating the process is its ability to validate user-defined integrity constraints over its execution. That is, whenever a new object/relation is created/deleted, OpExec ensures that no defined constraint is being violated, otherwise, the data update is rejected and a warning is returned to the client. For instance, when executing the *Change Vehicle Status* task, we might violate the condition that *damagedVehicles cannot be booked by a User*. If this is the case, *OpExec* notifies the client about this problem and rejects the execution of the task. Thus, the user might notice that the *User chooses and selects a car* task, requires selecting a car which is not only available, but also not damaged. Note that data inconsistencies might arise independently of the Service Blueprint lane in which the data update is performed, thus, they are not taken into account in our validation approach.

In order to ensure the efficiency of these checking, OpExec integrates an incremental checking approach [15], that is, it only checks those constraints that might be violated according to the data update, and only for the relevant values. It is worth mentioning that OpExec is implemented as a Java library that can be invoked from any other tool.

We plan then to integrate both OpExec and INNoVaServ as follows: model validation will rely on EVL scripts bundled in INNoVaServ, so that when such validation is run, the EVL rules invoke internally OpExec functions, which will then return the results that will be graphically displayed by INNoVaServ. Even though this process is slightly less efficient than simple EVL or OCL-based validation, it ensures not only syntactic but also semantic correctness.

4 Related Works

This section reviews existing works in the are of service blueprinting and process executability.

Even though service blueprinting emerged in the 80's [4], it has not attracted too much attention from academics until recently and most of the existing literature is focused on the application of the technique to different contexts. Regarding the combination with formal techniques, in [16] Berkley uses phase distributions to control service operations whereas fuzzy graph is used in [17] to modularize product extension service blueprints. There are also some works on the

combination of service blueprinting with the Theory of Inventive Problem Solving (TRIZ), like the one from Lee et al. [18]. As well, there are different works on the revision or extension of service blueprinting for specific purposes. For instance, Flieb and Kleinatelkamp presented a revised version of service blueprints in [19] based on the production-theoretic approach to identify starting points for improving process efficiency.

Regarding tool-support for service blueprinting, as the rise of product-service-systems [20] has contributed to increase the interest in this user-centered technique for business process modeling, most of existing works have emerged recently from the industry. This way, tools like Canvanaizer[9] and Real Time Board[10] to name a few are web-based applications that support collaborative edition of (canvas and) service blueprints. They bundle a simple and intuitive graphical interface (specially the latter) but, in contrast with *INNoVaServ* they were not devised to work with models, so they are limited to offer graphical representations of the blueprint, which can not be processed later.

From a more academic point of view, some remarkable works are those from Liang et al. [21], who use a CAD-based system for service blueprinting and the one from Lao [22] who developed a collaborative tabletop tool for service design based on some of the principles of service blueprinting. All in all, these are tools focused on usability and collaborative properties which have dismissed the utility of model-based tool support as a way to enable the systematic processing of the information collected in the blueprint. Thus they are very far from being ready to incorporate any kind of formal reasoning.

On the other hand, a quick look at the plenty of systematic literature reviews on business process modeling and the topics covered by them shows that this is somehow a most mature field. Recent reviews are indeed not focused on characterizing existing proposals, since that has been largely done in the past, but on available mechanisms to assess their quality [23] or complexity [24].

Regarding process executability, the approach in [14] uses a UML class diagram, a BPMN diagram and a set of OCL operation contracts to achieve process executability. Some of the advantages of [14] in contrast to our work are that it uses the *de-facto* standard modeling languages for data and processes, together with the fact that the OCL language has a more expressive power than structured natural language. Thus, it requires that the modeler and business people know BPMN and OCL, whereas service blueprints and structured natural language are simpler and more intuitive.

BPEL (or WS-BPEL) allows to specify executable business processes using an XML format which makes it difficult to read. Although there is a mapping between BPMN 2.0 and BPEL it is incomplete and suffers from several issues [25]. The approach in [26] uses XML nets, a Petri-net-based process modelling approach which is meant to be executable. It uses a graphical language, which maps to a DTD (XML Document Type Definition) to represent the data required by the process, and the data manipulations are graphically

[9] https://canvanizer.com/.
[10] https://realtimeboard.com/.

shown in the XML net. In contrast to our approach, this solution is technology-based, as the specification of the models is based on XML, and details of how to achieve executability are not explained.

YAWL [27] is a workflow graphical language whose semantics are formally defined and based on Petri nets, with its corresponding execution engine. Intuitively, the tasks are annotated with their inputs and outputs, without defining what changes are made by each of them. Thus, the execution engine only detects missing information and it is not able to fully execute the operation.

In [28] it is possible to obtain automatically an imperative model that is executable in a standard Business Process Management System. However, data is defined as a set of unstructured variables and the pre and postconditions merely state conditions over the data, instead of indicating exactly what is done by the different tasks.

Earlier attempts are [29,30]. Both approaches focus on defining a conceptual model which can then be automatically translated to achieve execution. However, the purpose of [29] is different to ours: their main goal is to be able to validate the model through execution, while ours is to achieve executability by using a combination of UML class diagram and service blueprint enriched with structured natural language. Similarly, the approach in [30] - which translates the models into Pascal - is outdated by object-oriented programming languages.

Finally, there are many different works that deal with verification and validation in business process models, such as [31,32]. However, these techniques do not execute the model as we do and, to the best of our knowledge, none of them use service blueprints.

To sum up, none of the analysed works rely on service blueprints as a way of modeling the business process. Moreover, not all of them provide the ability of executing the model automatically using a structured data model. Finally, none use structured natural language to specify the meaning of each of the tasks, thus requiring concrete knowledge of the language used to do so.

5 Conclusion and Further Work

This work has presented a framework for defining service blueprints that can be validated using simulation techniques. Moreover, we have proposed the implementation of the framework integrating two tools: *INNoVaServ*, which is model-based tool for service blueprinting, and *OpExec* which is a model simulator. The linkage is done by attaching semantic annotations to service blueprint tasks, and translating them into executable logic rules.

To the best of our knowledge, this is the first work that relies on service blueprinting as an executable business process modeling technique. Moreover, it does so in the context of a toolkit for business modeling that enables the development of bridges with other notations for business (process) like Canvas [33], e3Value [11], Process Chain Networks [12] or BPMN.

This paper addresses consequently one of those which has been acknowledged to be the main problems of service design: the lack of proper technical support [20]. The constant and rapid development of new services, products or product-service offerings to address new needs as soon as they appear is indeed a must for any organization, giving rise to an increasing interest in the discipline of service design. However, being an emerging field, this is one of those areas in which industry is ahead of academia, giving rise to the advent of solutions which does not always meet the desirable criteria in terms of quality.

The development of this type of proposals will help as well to mitigate the differences and challenges that emerge between different worlds that speak different languages, as it is the case with the variety of stakeholders typically involved in the development of digital products or services nowadays [34].

Acknowledgments. This research has been funded by the Ministry of Science and Innovation under the ELASTIC project (TIN2014-52938-C2-1-R), the Government of Madrid under the SICOMORo-CM project (S2013/ICE- 3006) and by the SSME Research Excellence Group (Ref. 30VCPIGI05) co-funded by URJC and Banco Santander.

References

1. Fisk, R.P., Brown, S.W., Bitner, M.J.: Tracking the evolution of the services marketing literature. J. Retail. **69**(1), 61–103 (1993)
2. Spohrer, J., Maglio, P.P., Bailey, J., Gruhl, D.: Steps toward a science of service systems. Computer **40**(1), 71–77 (2007)
3. Cook, L.S., Bowen, D.E., Chase, R.B., Dasu, S., Stewart, D.M., Tansik, D.A.: Human issues in service design. J. Oper. Manage. **20**(2), 159–174 (2002)
4. Shostack, G.L.: Designing services that deliver. Harvard Bus. Rev. **62**(1), 133–139 (1984)
5. Gounaris, S., Tanyeri, M., Kostopoulos, G., Gounaris, S., Boukis, A.: Service blueprinting effectiveness: drivers of success. Int. J. Manag. Serv. Qual. **22**(6), 580–591 (2012)
6. Van Gorp, P., Dijkman, R.: A visual token-based formalization of BPMN 2.0 based on in-place transformations. Inf. Softw. Technol. **55**(2), 365–394 (2013)
7. Noguera, M., Hurtado, M.V., Rodríguez, M.L., Chung, L., Garrido, J.L.: Ontology-driven analysis of UML-based collaborative processes using OWL-DL and CPN. Sci. Comput. Program. **75**(8), 726–760 (2010)
8. Gronback, R.C.: Eclipse Modeling Project: A Domain-specific Language (DSL) Toolkit. Pearson Education, London (2009)
9. Bitner, M.J., Ostrom, A.L., Morgan, F.N.: Service blueprinting: a practical technique for service innovation. Calif. Manag. Rev. **50**(3), 66–94 (2008)
10. Osterwalder, A., Pigneur, Y.: Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers. Wiley, Hoboken (2010)
11. Gordijn, J., Akkermans, H., Van Vliet, J.: Designing and evaluating e-business models. IEEE Intell. Syst. **16**(4), 11–17 (2001)
12. Sampson, S.E.: Visualizing service operations. J. Serv. Res. **15**(2), 182–198 (2012)

13. Vara, J.M., Marcos, E.: A framework for model-driven development of information systems: technical decisions and lessons learned. J. Syst. Softw. **85**(10), 2368–2384 (2012)
14. De Giacomo, G., Oriol, X., Estañol, M., Teniente, E.: Linking data and BPMN processes to achieve executable models. In: Dubois, E., Pohl, K. (eds.) CAiSE 2017. LNCS, vol. 10253, pp. 612–628. Springer, Cham (2017). doi:10.1007/978-3-319-59536-8_38
15. Oriol, X., Teniente, E., Rull, G.: TINTIN: a tool for incremental integrity checking of assertions in SQL server. In: Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, 15–16 March 2016, pp. 632–635 (2016)
16. Berkley, B.J.: Analyzing service blueprints using phase distributions. Eur. J. Oper. Res. **88**(1), 152–164 (1996)
17. Song, W., Wu, Z., Li, X., Xu, Z.: Modularizing product extension services: an approach based on modified service blueprint and fuzzy graph. Comput. Indust. Eng. **85**, 186–195 (2015)
18. Lee, C.H., Wang, Y.H., Trappey, A.J.: Service design for intelligent parking based on theory of inventive problem solving and service blueprint. Adv. Eng. Inform. **29**(3), 295–306 (2015)
19. FlieB, S., Kleinaltenkamp, M.: Blueprinting the service company. J. Bus. Res. **57**(4), 392–404 (2004). European Research in Service Marketing
20. Cavalieri, S., Pezzotta, G.: Product - service systems engineering: state of the art and research challenges. Comput. Ind. **63**(4), 278–288 (2012)
21. Liang, T.P., Wang, Y.W., Wu, P.J.: A system for service blueprint design. In: 2013 Fifth International Conference on Service Science and Innovation (ICSSI), pp. 252–253. IEEE (2013)
22. Lau, N.: ServiceSketch: a collaborative tabletop tool for service design (2011)
23. de Oca, I.M.M., Snoeck, M., Reijers, H.A., Rodriguez-Morffi, A.: A systematic literature review of studies on business process modeling quality. Inf. Softw. Technol. **58**, 187–205 (2015)
24. Polančič, G., Cegnar, B.: Complexity metrics for process models - a systematic literature review. Comput. Stand. Interfaces **51**, 104–117 (2017)
25. Fabra, J., de Castro, V., Álvarez, P., Marcos, E.: Automatic execution of business process models: exploiting the benefits of model-driven engineering approaches. J. Syst. Softw. **85**(3), 607–625 (2012)
26. Lenz, K., Oberweis, A.: Modeling interorganizational workflows with XML nets. In: HICSS-34. IEEE Computer Society (2001)
27. Foundation, T.Y.: YAWL - User Manual. Version 4.1. (2016). http://www.yawlfoundation.org/pages/support/manuals.html
28. Parody, L., López, M.T.G., Gasca, R.M.: Hybrid business process modeling for the optimization of outcome data. Inf. Softw. Technol. **70**, 140–154 (2016)
29. Lindland, O.I., Krogstie, J.: Validating conceptual models by transformational prototyping. In: Rolland, C., Bodart, F., Cauvet, C. (eds.) CAiSE 1993. LNCS, vol. 685, pp. 165–183. Springer, Heidelberg (1993). doi:10.1007/3-540-56777-1_9
30. Mylopoulos, J., Borgida, A., Greenspan, S.J., Wong, H.K.T.: Information system design at the conceptual level - the taxis project. IEEE Database Eng. Bull. **7**(4), 4–9 (1984)

31. Gonzalez, P., Griesmayer, A., Lomuscio, A.: Verification of GSM-based artifact-centric systems by predicate abstraction. In: Barros, A., Grigori, D., Narendra, N.C., Dam, H.K. (eds.) ICSOC 2015. LNCS, vol. 9435, pp. 253–268. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48616-0_16
32. Deutsch, A., Hull, R., Vianu, V.: Automatic verification of database-centric systems. SIGMOD Rec. **43**(3), 5–17 (2014)
33. Ovans, A.: What is a business model. Harvard Bus. Rev. **23** (2015)
34. Gray, J., Rumpe, B.: Models for the digital transformation. Softw. Syst. Model. **16**(2), 1–2 (2017)