SOCA (2017) 11:445–458
https://doi.org/10.1007/s11761-017-0220-2

SPECIAL ISSUE PAPER

Cloud service brokerage: enhancing resilience in virtual enterprises through service governance and quality assurance

Simeon Veloudis[1] · Iraklis Paraskakis[1] · Christos Petsos[1]

Received: 1 February 2017 / Revised: 11 July 2017 / Accepted: 9 October 2017 / Published online: 24 October 2017

Abstract We argue that cloud service brokerage (CSB) mechanisms can strengthen the resilience of services in cloud-based VEs. In this respect, we present the Service Completeness-Compliance Checker (SC^3), a mechanism which offers capabilities with respect to the quality assurance dimension of CSB. More specifically, the SC^3 strengthens the resilience of cloud services by evaluating their conformance with pre-specified policies concerning the business aspects of their delivery, as well as by managing the evolution of their lifecycle in a controlled and policy-based manner. By relying on an ontology-based representation of policies and services, the SC^3 achieves a proper separation of concerns between policy definition and policy enforcement. This effectively enables the SC^3 to operate in a manner generic and agnostic to any underlying cloud delivery platform, as well as to reason about the well-formedness of the pre-specified policies.

Keywords Virtual enterprises · Cloud computing · Cloud service brokerage · Governance · Ontologies

✉ Simeon Veloudis
sveloudis@seerc.org

Iraklis Paraskakis
iparaskakis@seerc.org

Christos Petsos
chpetsos@seerc.org

1 South East European Research Centre (SEERC), The University of Sheffield, International Faculty CITY College, 24 Proxenou Koromila St, 54622 Thessaloníki, Greece

1 Introduction

Cloud computing has evolved out of Grid computing [10,26] as a result of a shift in focus from an infrastructure aiming to deliver mainly storage and compute resources, to an economy-based computing paradigm aiming to deliver a wide range of resources abstracted as services [10]. Such a shift is anticipated to impact the manner in which businesses and organisations share skills and core competencies within a distributed collaborative network [28]. More specifically, activities performed by a dynamic multi-institutional virtual enterprise (VE) may involve the use of heterogeneous, externally sourced cloud services which span different clouds and capability levels (IaaS, PaaS, and SaaS) [5], and which are entrusted by their users with data, software, and computation; we shall term such a VE a cloud-based one.

As an example, consider the following scenario. An industrial consortium is formed to collaboratively process the data produced as part of a seismic survey. Such processing integrates software components, offered as a service, by different consortium participants (SaaS offerings). Each component may be operating on a participant's proprietary infrastructure or, alternatively, on infrastructure provisioned as a cloud service (IaaS offering). At the same time, the simulation requires the development of new specialised software components. To this end, the consortium is provisioned the necessary software platform for developing these applications as a service (PaaS offering).

Evidently, the IT environment of a cloud-based VE is transformed into a complex ecosystem of intertwined infrastructure, platform, and application services delivered remotely, over the Internet, by diverse service providers. As the number of services proliferates, it becomes increasingly difficult to keep track of when and how these services evolve over time, either through intentional changes, initiated by their providers, or through unintentional changes, such as variations in their performance and availability. It is therefore increasingly important to instil into these services *resilience* aspects both at design time and at run-time.

According to [9], resilience is the capacity to survive, adapt, and grow in the face of turbulent change. In our case, for the survival and adaptation of cloud services we require, for example, that at design time cloud services are provided redundantly, e.g. through at least two distinct endpoints. Additionally, for survival and adaptation at run-time, we require the existence of mechanisms capable of predicting and evading potential failure of a service. Should a failure occur, the mechanism must be able to recover from that failure and enable the smooth operation of the service that contributes to the overall business continuity and growth.

In order to strengthen the resilience of the services in such an ecosystem, cloud-based VEs are anticipated to increasingly rely on cloud service brokerage (CSB) [16]. In this respect, the work in [28] proposed a conceptual architecture of a framework which offers capabilities with respect to two dimensions of CSB, namely Quality Assurance Service Brokerage and Service Customisation Brokerage. These capabilities revolve around the following themes: (i) governance and quality control; (ii) failure prevention and recovery; and (iii) optimisation. The first theme strengthens the resilience of cloud services by checking their compliance with pre-specified policies concerning the technical, business, and legal aspects of service delivery; it also strengthens the resilience of cloud services by testing their conformance with their expected behaviour and by continuously monitoring their operation. The second theme strengthens the resilience of cloud services through the reactive and proactive detection of service failures, and the selection of suitable strategies to prevent—or recover—from such failures. Finally, the third theme ensures that an acceptable level of service is maintained in the face of changing user requirements—this is achieved by continuously optimising service consumption with respect to a diverse set of quantitative and qualitative characteristics such as cost, quality, and functionality.

Continuing the work in [28], this paper reports on the implementation of a particular mechanism of the aforementioned framework, namely the Service Completeness-Compliance Checker (SC^3), which offers capabilities with respect to the governance and quality control theme. More specifically, the SC^3 strengthens the resilience of cloud services by:

- evaluating their conformance with pre-specified policies concerning their business aspects of delivery;
- managing the evolution of their lifecycle in a controlled and policy-based manner.

By relying on a declarative representation of policies and services, one which is based on an RDF(S) [22] ontology, the SC^3 achieves a clear separation of concerns: policies are represented at a higher level of abstraction, independently of the code that the SC^3 employs for enforcing them. This brings about the following seminal advantages. (i) It keeps the SC^3 generic and orthogonal to any underlying cloud delivery platform employed by a cloud-based VE. (ii) It allows reasoning about the well-formedness, hence the effectiveness, of the policies. It is to be noted here that such a separation of concerns is generally absent in contemporary governance mechanisms [27], with negative repercussions on their portability, reusability, as well as on their ability to automatically reason about the effectiveness of the policies that they employ.

The rest of this paper is structured as follows. Section 2 presents a motivating scenario. Section 3 outlines our declarative approach to policy representation. Section 4 outlines how the SC^3 extracts from this representation the necessary information for the subsequent service and policy evaluation processes. Section 5 describes how the SC^3 evaluates the conformance of services with policies, and Sect. 6 describes how the SC^3 evaluates the well-formedness of these policies. Section 7 briefly elaborates on how the SC^3 manages the lifecycle of services in a policy-based manner, and Sect. 8 outlines our approach to testing the SC^3. Finally, Sect. 9 presents related work and Sect. 10 outlines conclusions and future work.

2 Motivating scenario

The work reported in [28] proposed a conceptual architecture of a CSB framework offering capabilities spanning the main phases of a service's lifecycle, namely Service On-boarding, Service Operation, and Service Evolution. The SC^3 offers capabilities with respect to the *Service On-boarding* and *Service Evolution* phases.[1] Below, we identify these capabilities through the prism of the example of Sect. 1.

Let CPx (stands for Cloud Platform x) be a cloud delivery platform that hosts various services that are potentially used by the industrial consortium. The platform houses a variety of apps developed by CPx's network of ecosystem partners. CPx also allows advanced users to develop and deploy custom applications on the platform and to create rich compositions of applications (mashups) offered by third-party service providers.

Suppose that an ecosystem partner offers a new service on CPx, call it StoreCloud, which provides an encrypted and

[1] The SC^3 also offers capabilities with respect to the Service Operation phase and, in particular, with respect to continuously monitoring the behaviour of a service. These capabilities shall not, however, concern us in this paper.

Table 1 Entry-level criteria

Service-level attribute	Acceptable values	SLO	Description
Storage	[100, 1000)	Gold	Size (TB)
	[10, 100)	Silver	
	[0, 10)	Bronze	
Availability	[0.99999, 1)	Gold	Uptime ratio
	[0.9999, 1)	Silver	
	[0.999, 1)	Bronze	
Encryption	256	Gold	Key length
	192	Silver	
	128	Bronze	

versioned persistence layer for storing intermediate results during the various phases of the seismic survey. In order for the new service to be on-boarded on CPx, a number of entry-level criteria must be satisfied. These crucially capture a set of *service-level objectives* (SLOs) expressed in terms of restrictions on relevant *service-level attributes*. Table 1 summarises the service-level attributes, and their corresponding SLOs, considered for the purposes of this example. These SLOs essentially form CPx's *business* (or *broker*[2]) *policy* (BP) with respect to on-boarding StoreCloud.

We assume that the ecosystem partner who offers StoreCloud, hereafter referred to as the *service provider* (SP), submits a service description (SD) which details the manner in which StoreCloud is to be deployed on CPx. The SC^3 offers an *SD evaluation capability* which essentially allows the cloud-based VE to determine whether this SD is compliant with CPx's BP. Such a capability entails two kinds of evaluation: SD *completeness* evaluation and SD *compliance* evaluation. The former kind of evaluation aims at determining whether the SD specifies values for all required service-level attributes. For example, an SD which does not specify a value for the encryption attribute cannot be considered complete. The latter kind of evaluation aims at determining whether the specified attribute values fall within the corresponding ranges prescribed in the BP. For example, an SD which specifies a 64-bit value for the encryption attribute cannot be considered compliant. Clearly, the aforementioned evaluations seek to determine whether StoreCloud attains the SLOs specified in CPx's BP. In this respect, they strengthen the service's resilience.

The SC^3 also offers a *policy evaluation capability* and a *service lifecycle management capability*. The former essentially allows the cloud-based VE to determine the *well-formedness*, hence the effectiveness, of CPx's BP. The latter ensures that services evolve, e.g. through updates and deprecation activities, in a controlled and policy-based manner.

The aforementioned capabilities reflect—and therefore hinge upon—our *declarative framework* for representing the SLOs incorporated in the BP and, ultimately, the BP itself. In this respect, it is essential that we provide an account of this declarative framework before proceeding to explain how the SC^3 in fact offers these capabilities.

3 Declarative representation of BPs

As already mentioned in Sect. 1, our declarative framework for the representation of BPs and SDs unravels the definition of a policy from the actual code that the SC^3 employs for enforcing it. This brings about the following seminal advantages: (i) it keeps the SC^3 generic and orthogonal to the underlying cloud delivery platform as policies are expressed in a higher-level formalism; (ii) it forms an adequate basis for reasoning generically about the well-formedness, hence the effectiveness, of the broker policies. Moreover, it enables the identification of inter-policy relations such as subsumption and contradiction and facilitates the overall governance of policies.

The declarative framework is based upon Linked USDL [15]—a lightweight RDF(S) vocabulary of concepts and properties for modelling, comparing, and trading services and service bundles, as well as for specifying, tracking, and reasoning about the involvement of entities in service delivery chains. Sections 3.1 and 3.2 below provide an account of this declarative framework; an outline of the advantages offered by the adoption of Linked USDL is, however, first in order.

Firstly, Linked USDL draws upon a number of widely adopted vocabularies such as GoodRelations [11], SKOS [23], and FOAF [24]. It therefore promotes knowledge sharing, whilst it increases the interoperability, and thus the reusability and generality, of broker policies. Secondly, by embracing Linked Data as the core means for capturing facts about people, organisations, resources and services, Linked USDL supports large-scale, efficient, multi-party interactions through linking to other ontologies [3,21]. Thirdly, by offering a number of different profiles,[3] Linked USDL provides a holistic and generic solution able to adequately capture a wide range of business details.

These advantages facilitate the evaluation of the conformance of cloud services with pre-specified policies concerning their business aspects of delivery and deployment, i.e. with policies that ensure that services indeed meet certain objectives regarding the level of functionality that they are expected to provide. In this respect, they are significant from

[2] We use the term 'broker' to emphasise that, in our work, such a business policy is formulated according to the declarative approach of our *brokerage* framework (see Sect. 3).

[3] Such as SLA, Security, IPR, Pricing [15].

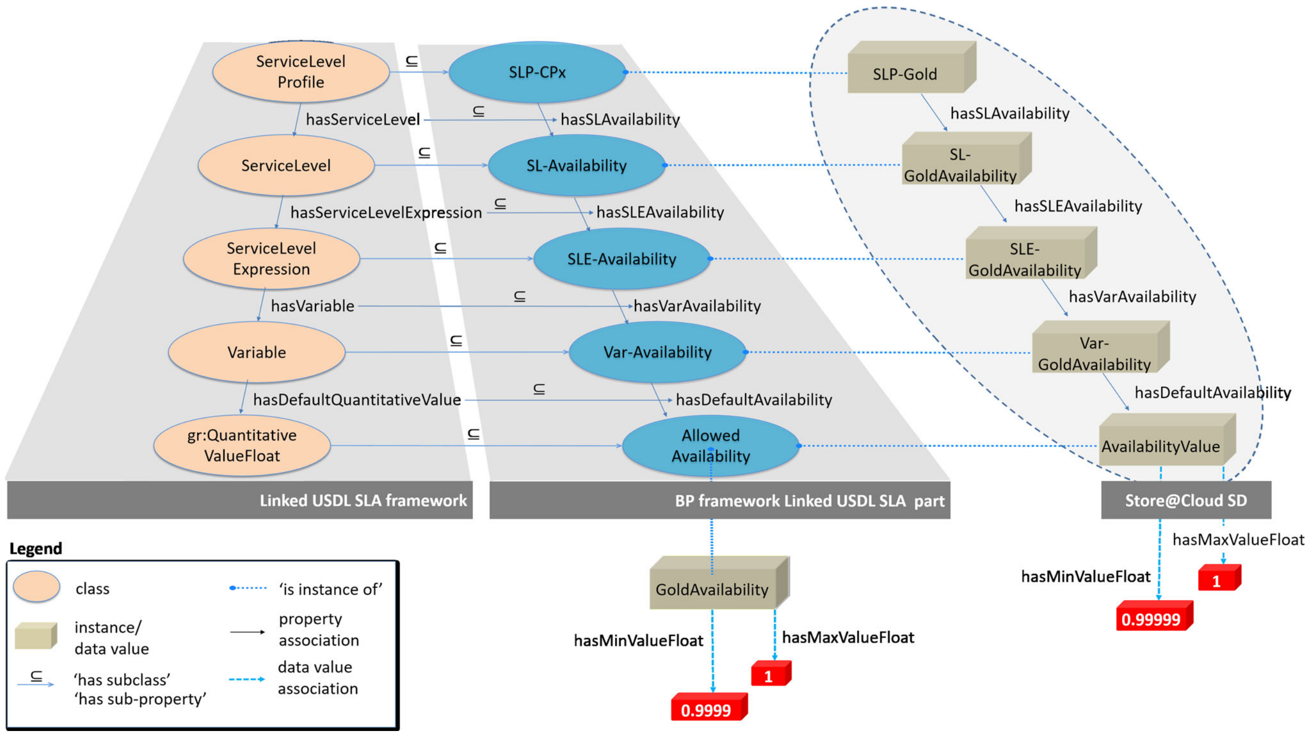

Fig. 1 Linked USDL SLA for the 'gold' availability SLO

the standpoint of increasing our assurance upon the resilience of cloud services. For instance, a policy may require that a particular service is available from at least two distinct endpoints, hence increasing our assurance upon the resilience of the service in case one of the endpoints goes down.

Linked USDL comprises a Core schema which in our model is used for representing BPs as well as for encoding certain invariable characteristics of BPs (see Sect. 3.2 for more details). From this Core schema, a number of extension schemata hinge addressing diverse business aspects of a BP (such as Pricing, SLA, Security, and IPR); for the purposes of this work, we focus on the SLA schema. In particular, Sect. 3.1 provides an account of how our declarative framework for the representation of policies is derived as a specialisation of Linked USDL SLA.

3.1 Declarative representation of BPs and SLOs in Linked USDL SLA

We model the SLOs of a BP, hence the BP itself, through a *specialisation process* which constructs a framework of suitable subclasses and sub-properties of the Linked USDL SLA classes and properties depicted in Fig. 1.[4] These subclasses are then populated by instances specified in the SDs that are evaluated for conformance with the BP (e.g. StoreCloud's SD). Below, we outline this process for the 'gold' SLO of StoreCloud's availability attribute[5] (see Sect. 2).

3.1.1 SLO representation

For each service-level attribute, the BP offers a subclass of the class ServiceLevel for accommodating the attribute's SLOs. For example, for accommodating the SLOs of the availability attribute (i.e. the 'gold', 'silver', and 'bronze' SLOs of Table 1), it offers the class SL-Availability (see Fig. 1). SLOs appear as instances of this class—e.g. the SL-GoldAvailability instance specified in StoreCloud's SD (see Fig. 1).

Each SLO is defined in terms of a *service-level expression* (SLE) which specifies the conditions that must be satisfied in order for the SLO to be met. More specifically, for each service-level attribute, the BP offers a subclass of the class ServiceLevelExpression for accommodating the

[4] Note that in order to reduce notational clutter we avoid specifying namespaces for classes and properties, unless a class or property comes from an external ontology (e.g. the GoodRelations ontology). In addition, the following conventions are used in the figures of this paper (see

Footnote 4 continued
also the legend of Fig. 1): a class is represented by an oval; a property is represented by an arrow decorated with the name of the property; a subclass relation is represented by an arrow decorated with the subset symbol (⊆); instance-class associations are represented with perforated lines.

[5] Of course, an analogous account applies to the rest of the attributes and SLOs of Table 1.

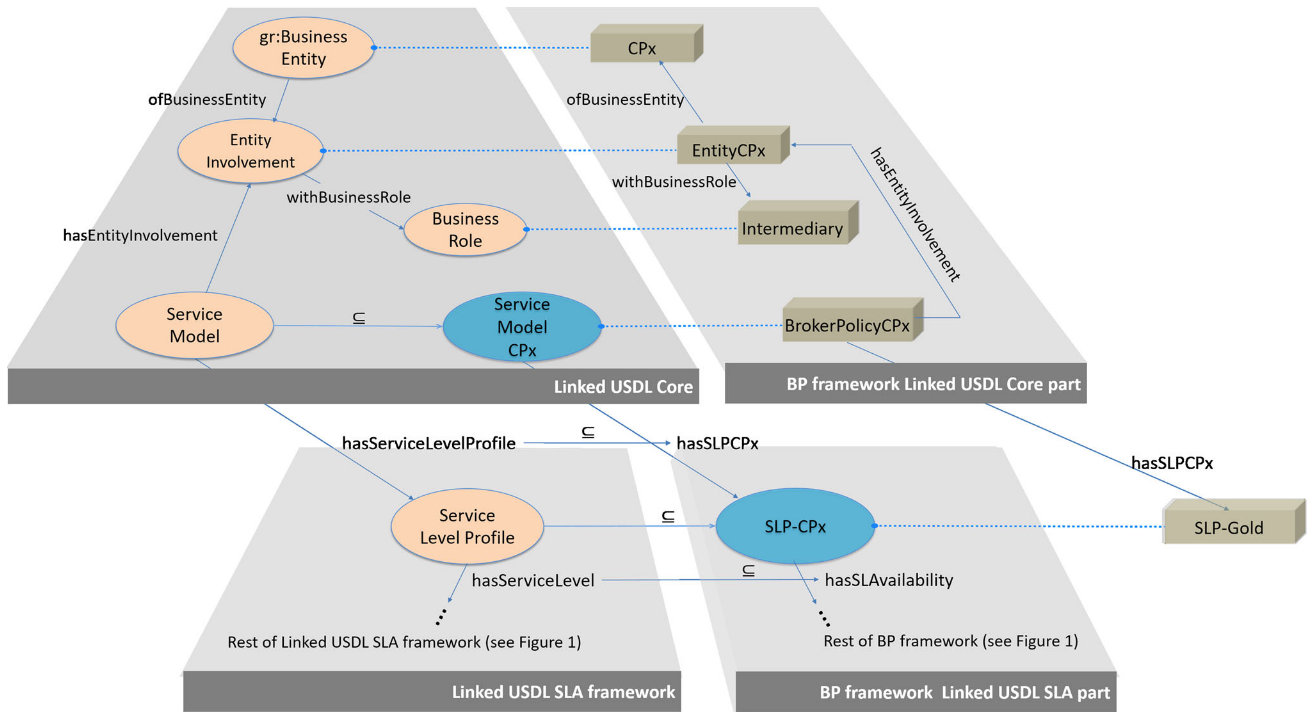

Fig. 2 Linked USDL Core classes, interrelations and instances

SLEs that correspond to that attribute's SLOs. For example, the SLEs that correspond to the availability attribute's SLOs are modelled as instances of the SLE-Availability subclass (see Fig. 1). These instances appear in the SDs that are evaluated for conformance with the BP—e.g. the SLE-GoldAvailability instance in StoreCloud's SD. SLOs are associated with their corresponding SLEs through appropriate sub-properties of the hasServiceLevelExpression property. In particular, the SLOs related to the availability attribute are associated with their SLEs through the hasSLEAvailability property (see Fig. 1).

Each SLE binds a *variable* that corresponds to a particular attribute, one which is associated with an allowable range of values. Following an approach entirely symmetrical to the one outlined above for SLOs and SLEs, variables are modelled as instances of appropriate subclasses of the class Variable (e.g. the class Var-Availability depicted in Fig. 1), whilst value ranges are modelled as instances of appropriate subclasses of the GoodRelations class gr:QuantitativeValueFloat[6] (e.g. the class AllowedAvailability depicted in Fig. 1). These instances appear in the SDs that are evaluated for conformance with the BP—e.g. the Var-GoldAvailability and AvailabilityValue instances in StoreCloud's SD. SLEs are associated with their corresponding variables through sub-properties of the hasVariable property (e.g. the hasVarAvailability of Fig. 1), and variables are associated with their allowable values through sub-properties of the hasDefaultQuantitativeValue property.[7]

3.1.2 *Service-level profiles*

Section 3.1.1 outlined a framework for representing SLOs. This framework is associated with the pertinent BP[8] through the concept of a *service-level profile* (SLP). SLPs are essentially groupings of SLOs whose purpose is to formulate different service deployment 'packages' that are allowable by the BP. For example, in the scenario of Sect. 2, the 'gold' SLP formulates a deployment package comprising the 'gold' SLOs of the attributes of Table 1.[9]

SLPs take the form of instances of appropriate subclasses of the ServiceLevelProfile class, e.g. the SLP-Gold instance of the SLP-CPx subclass of Fig. 1. SLPs are associated with their constituent SLOs through appropriate sub-properties of the hasServiceLevel property—an

[6] Or of the class gr:QualitativeValue, in case of qualitative variables.

[7] Or through sub-properties of the hasDefaultQualitativeValue, in case of qualitative values.

[8] Recall from Sect. 2 that a BP is, after all, a set of SLOs.

[9] Of course, which SLOs are comprised by a particular SLP is an application-specific issue determined by CPx itself. For instance, CPx may choose to define a 'gold' SLP as comprising either 'gold'-only SLOs, or two 'gold' SLOs and a 'silver' SLO; alternatively, it may choose to define the latter grouping as a 'silver' SLP.

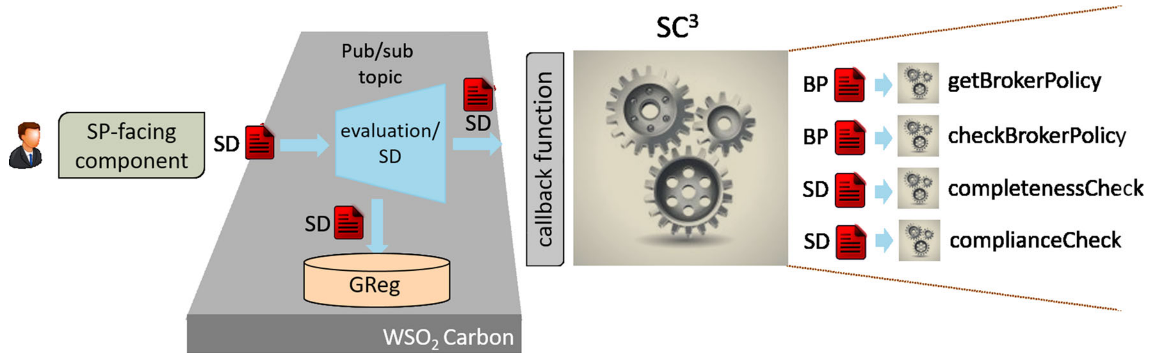

Fig. 3 Conceptual architecture

example is the property hasSLAvailability of Fig. 1 which associates the 'gold' SLP with the 'gold' availability SLO.

We demonstrate next how BPs are linked with their SLPs and thus ultimately with the SLOs that they offer. To this end, Linked USDLs Core schema is utilised.

3.2 Declarative representation of BPs in Linked USDL core

A BP is represented as an instance of the Linked USDL Core class ServiceModel (see Fig. 2). More precisely, it takes the form of an instance of a subclass of this class—e.g. the instance BrokerPolicyCPx of the class ServiceModel-CPx depicted in Fig. 2.[10] The purpose of such a subclass is to accommodate all of CPx's BPs.[11] A BP is associated with the SLPs that it encompasses through sub-properties of the Linked USDL Core property hasServiceLevelProfile—e.g. through the hasSLPCPx property of Fig. 2.

In addition to modelling BPs, Linked USDL's Core schema has an additional role to play: it provides an adequate basis for formally capturing certain tangential knowledge artefacts about a BP[12]—these are: (i) the identity of the business entity which is responsible for defining the BP (i.e. CPx, in the case of the scenario of Sect. 2); (ii) the role in the capacity of which this business entity acts when defining the BP. These knowledge artefacts are associated with the BP through the Linked USDL classes and properties depicted in Fig. 2. More specifically, BrokerPolicyCPx is associated with an instance, say EntityCPx, of the class EntityInlvolvement through the property hasEntityInvolvement. EntityCPx is further associated with an instance, say CPx, of the class gr:BusinessEntity via the property ofBusinessEntity. The instance CPx identifies the business entity which is responsible for defining the BP. In addition, EntityCPx is associated with the instance Intermediary of the class gr:BusinessRole via the property withBusinessRole. The instance Intermediary identifies the role in the capacity of which CPx acts when defining the BP.

[10] The classes and properties depicted in Fig. 2 are by no means the complete set of classes and properties offered by Linked USDL Core, but rather an appropriate subset discerned for the purposes of this work.

[11] Although in this paper we concentrate (without loss of generality) on a single BP, a cloud delivery platform such as CPx may employ a number of different policies—e.g. in order to accommodate the needs of different service categories.

[12] Tangential in the sense that they do not describe core aspects of a BP but rather focus on pertinent peripheral information.

4 SC^3: BP parsing

This section describes how the SC^3 parses a BP in order to extract the necessary information for evaluating the completeness and compliance of SDs, as well as for evaluating the well-formedness of the BP. A brief description of a conceptual architecture for the SC^3 is, however, first in order.

4.1 Conceptual architecture

As depicted in Fig. 3, the SP submits StoreCloud's SD through the SP-facing component—an interface which exposes an editor for facilitating the construction of the SD. The SD is then transported to the SC^3 and also stored in the Governance Registry (GReg) depicted in Fig. 3; the transportation takes place through a Publish/subscribe (Pub/sub) system. An explanation of the reasons for opting for the open-source WSO$_2$ Carbon platform [36] (see Fig. 3), as well as for advocating a Pub/sub system for transporting SDs, is omitted here; a relevant discussion can be found in [6,7].

The SC^3 exposes a mechanism which allows subscribers to register a callback function for the appropriate topic of the Pub/sub system and (asynchronously) receive the SD. More specifically, this mechanism utilises the Evaluation-ComponentSDSubscriber class which is responsible for orchestrating all the actions required for creating connections to the Pub/sub system and subscribing to its topics. This

Table 2 HashMap objects

```
Map<String, BrokerPolicyClass> serviceModelMap;
Map<String, BrokerPolicyClass> serviceLevelProfileMap;
Map<String, BrokerPolicyClass> serviceLevelMap;
Map<String, BrokerPolicyClass> serviceLevelExprMap;
Map<String, BrokerPolicyClass> expressionVarMap;
Map<String, BrokerPolicyClass> quantValFloatMap;
```

class triggers the SC^3 when a fresh SD arrives. In particular, it invokes an object of the class PolicyCompletenessCompliance, one which is parameterised with the appropriate pre-specified BP against which the evaluation will take place.[13] The PolicyCompletenessCompliance class is one of the core classes of the SC^3. It offers three main methods: getBrokerPolicy, completenessCheck, and complianceCheck. The first method extracts all the required information from the BP; on the basis of this information, the second and third methods determine the completeness and compliance of the SD with respect to the BP (see Sect. 5). The SC^3 also offers the method checkBrokerPolicy which evaluates the well-formedness of the (pre-specified) BP (see Sect. 6).

The aforementioned methods are implemented in Java using the Apache Jena (Core and ARQ) APIs [1]. In addition, as we would expect, all methods reflect—and therefore hinge upon—our declarative framework for the representation of BPs outlined in Sect. 3.

4.2 The getBrokerPolicy method

The getBrokerPolicy process parses the BP and places the information that it extracts in the bp object of the class BrokerPolicy. This class encompasses a number of Java HashMap objects as attributes (see Table 2). The HashMap objects reflect our declarative framework for representing the SLOs incorporated in the BP and, effectively, the BP itself. More specifically, the getBrokerPolicy method sets out to construct a programmatic (in-memory) representation of the BP framework outlined in Sect. 3. In this respect, it starts off by discovering, for each Linked USDL SLA class C, those subclasses S of C that appear in the BP. It then instantiates the HashMap attributes of the bp object (see Table 2) with the corresponding subclasses. This instantiation takes place through the method getBrokerPolicyClassMap as indicated by the following line of code for the ServiceLevel class (the rest of the instantiations are entirely analogous and thus omitted).

```
bp.setServiceLevelMap(
getBrokerPolicyClassMap(USDL_SLA,"ServiceLevel"))
```
(1)

The getBrokerPolicy method then proceeds to construct a list of string objects holding the URIs of all gr:QuantitativeValueFloat instances found in the BP (e.g. the GoldAvailability instance depicted in Fig. 1); pseudocode that populates this list is shown in Algorithm 1.

Algorithm 1 Discovering and programmatically representing QuantitativeValueFloat instances

```
1: floatQVs[] {declare empty array to hold QuantitativeValueFloat
   instance URIs}
2: for each <value, resourceURI> do
3:    if (typeOfValue = float) then
4:       floatQVs.add(resourceURI) {append to array}
5:    end if
6: end for
```

Subsequently, the getBrokerPolicy method proceeds to discover all properties in the BP, along with their corresponding ranges, which have as a domain one of the subclasses S; these are effectively all the sub-properties that appear in the BP. For each discovered sub-property, an object of the class Subproperty is constructed (see Algorithm 2 for an excerpt of the relevant code).

Algorithm 2 Discovering and programmatically representing sub-properties in the BP

```
Require: Properties {pre-populated array with parsed URIs that cor-
   respond to BP properties}
1: for each propertyURI in Properties do
2:    Subproperty p {declare sub-property construct}
3:    p.uri ← propertyURI
4:    p.domain ← extractDomainOf(propertyURI)
5:    p.range ← extractRangeOf(propertyURI)
6: end for
```

5 SC^3: SD evaluation

An SD is effectively a set of interconnected instances that populate the subclasses of the BP framework (see Fig. 1). Below, we outline the processes that evaluate an SD with respect to the BP. These processes essentially determine whether an SD is *complete* and *compliant* with respect to the BP. As already mentioned in Sect. 2, an SD is considered complete with respect to the BP when it specifies values for all required (according to the BP) service-level attributes.

[13] The BP is constructed through an interface that exposes an appropriate editor and is transported to the SC^3 through a relevant topic of the Pub/sub system (not shown in Fig. 3 to avoid clutter).

An SD is considered compliant with respect to the BP when the specified attribute values fall within the corresponding ranges prescribed by the BP.

5.1 The completenessCheck method

The completenessCheck algorithm starts off by determining whether the SD encompasses an instance I_S of each class S discovered by the getBrokerPolicy method. Then, for each object property P_o discovered by getBrokerPolicy such that $\text{dom}(P_o) = S$, it determines whether I_S is associated (via P_o) with exactly one instance $I_{S'}$ of the class $S' = \text{ran}(P_o)$. Similarly, for each data property P_d, such that $\text{dom}(P_d) = S$, it determines whether I_S is associated via P_d with a data value from $\text{ran}(P_d)$. Pseudocode that checks these associations is shown in Algorithm 3. For example, let S = SL-Availability (see Fig. 1). The algorithm initially checks whether the SD contains an instance of S (in this case SL-GoldAvailability). Let now P_o = hasSLEAvailability. The algorithm checks whether SL-GoldAvailability is associated, via P_o, with an instance of SLE-Availability (an association which exists with SLE-GoldAvailability). Analogous checks are performed for the rest of the instances in the SD framework.

Algorithm 3 Checking instance associations in SDs

```
1: Property P {declare construct P for holding property}
2: for each instance I do
3:     numOfAssocations ← countAssociations(I,P) {count number
       of associations of I via P}
4:     if (numOfAssociations != 1) then
5:         throw Exception
6:     end if
7: end for
```

5.2 The complianceCheck method

The compliance checking algorithm proceeds by determining whether the values, or value ranges, specified in the SD are in accordance with the allowable values, or value ranges, specified in the BP. More specifically, the algorithm starts off by determining the values that are associated with a gr:QuantitativeValueFloat instance (e.g. the instance Var-GoldAvailability of Fig. 1) via each of the properties hasMinValueFloat and hasMaxValueFloat. If these values are undefined, an error message is emitted (see Algorithm 4). Otherwise, the algorithm proceeds to check that the range associated with the gr:QuantitativeValueFloat instance is indeed subsumed by the corresponding range declared in the BP (see Algorithm 5). For example, the algorithm checks whether the range [0.99999, 1) associated with Var-GoldAvailability is subsumed by the range [0.9999, 1) associated with the corresponding AvailabilityValue instance in the BP (see Fig. 1).

Algorithm 4 Checking the value range associated with each instance in an SD

```
1: Instance I {declare construct I for holding instance}
2: minValue ← extractMinValue(I)
3: maxValue ← extractMaxValue(I)
4: if (minValue = null) $OR$ (maxValue = null) then
5:     throw Exception
6: end if
```

Algorithm 5 Checking subsumption between SD and BP value ranges

```
Require: valueRangesFromBP {pre-populated array with value
    ranges from BP}
1: for each valueRange in valueRangesFromBP do
2:     if (maxValue $<$ valueRange.max) $AND$ (minValue $\geq$ val-
       ueRange.min) then
3:         return Success
4:     end if
5: end for
```

6 SC^3: BP evaluation—the checkBrokerPolicy method

We now turn our attention to describing the evaluation mechanism for checking the *well-formedness* of a BP with respect to the ontological framework of Sect. 3. The mechanism is conceptually divided into two parts: one for evaluating the Linked USDL SLA portion of a BP and one for evaluating the Linked USDL Core portion of a BP. These parts are elaborated in Sects. 6.1 and 6.2.

Algorithm 6 Checking associations between SLOs and SLEs

```
Require: SLOs {pre-populated array with SLOs from BP}
1: for each SLO in SLOs do
2:     numOfSLEAssociations ← countSLEAssociations(SLO)
3:     if (numOfSLEAssociations $<$ 1) then
4:         throw Exception
5:     end if
6: end for
```

6.1 Evaluating the SLA portion of a BP

The evaluation mechanism performs a series of checks that are analogous to the ones outlined in Sect. 5.1 but which now concentrate on the BP framework (see Fig. 1) rather than on the interconnected instances that make up the SD. More specifically, the mechanism checks that each subclass that accommodates the SLOs of a particular attribute (i.e. each subclass of the class ServiceLevel) is associated with a subclass that accommodates the SLEs that correspond to that attribute's SLOs (i.e. to a subclass of the class ServiceLevelExpression—see Algorithm 6). It also checks that this association is materialised through a distinct sub-property of the property hasServiceLevelExpression (e.g. the property hasSLEAvailability of Fig. 1). An entirely symmetrical set of checks applies to the associations between SLEs and variables, as well as between variables and quantitative (or qualitative) values. Additionally, type-safety checks are performed for the declared values (see Algorithm 7).

Moreover, the evaluation mechanism checks that: (i) all quantitative value instances are delimited by minimum and maximum values through the data properties gr:hasMinValueFloat and gr:hasMaxValueFloat respectively; (ii) all quantitative value instances are assigned an appropriate type through the data property gr:hasUnitOfMeasurement .

Algorithm 7 Checking associations between variables and floats

```
Require: Variable V {a variable from BP}
1: value ← extractValueOf(V)
2: if (typeOfValue != float) then
3:     throw Exception
4: end if
```

The evaluation mechanism also checks that each subclass of the class ServiceLevelProfile is associated with one or more subclasses of the class ServiceLevel through distinct sub-properties of the property hasServiceLevel. For example, it checks that the SLP-CPx subclass of Fig. 1 that is specifically devised for accommodating all of CPx's SLPs is associated with the class SL-Availability through a sub-property of hasServiceLevel. Of course, an analogous check applies to any other subclasses of the class ServiceLevel that accommodate SLOs of other attributes (e.g. the SLOs of the storage and encryption attributes of Table 1).

Finally, the evaluation mechanism checks that there exists an association between the class ServiceModel (which accommodates the instance that represents the BP—see Fig. 2) and at least one subclass of the class ServiceLevelProfile. It also checks that this association is materialised through a sub-property of the property hasServiceLevelProfile (e.g. the hasSLPCPx property depicted in Fig. 2).

6.2 Evaluating the core portion of a BP

Our aim is to ascertain the following two facts: (i) there exists an instance of the class ServiceModel which identifies the BP; (ii) the Core portion of a BP captures the tangential knowledge mentioned in Sect. 3.2, namely the business entity responsible for defining the BP, as well as the role in the capacity of which this business entity acts when defining the BP. Concerning the former fact, the policy evaluation mechanism obtains all those resources that are defined as subclasses of the class ServiceModel, as well as all the instances that are encompassed in these subclasses. The mechanism then ensures that there exists exactly one instance in the subclass ServiceModelCPx (i.e. the BrokerPolicyCPx instance of Fig. 2) which identifies the BP.

Concerning the latter fact, the policy evaluation mechanism checks that: (i) BrokerPolicyCPx is associated with exactly one instance of the class EntityInvolvement via the hasEntityInvolvement property; (ii) this EntityInvolvement instance is associated with the instance CPx of the class gr:BusinessEntity, and with no other instances from that class; (iii) the same EntityInvolvement instance is associated with the instance Intermediary of the class BusinessRole, and with no other instances from that class.

7 SC^3: service lifecycle management

The SC^3 provides the following two functionalities with respect to service lifecycle management: (i) service updates evaluation; (ii) policy-based service deprecation and removal management. Regarding the former functionality, the SC^3 is responsible for ensuring the quality of any service updates[14] by evaluating their conformance with the BP. As we would expect, such an evaluation takes place in precisely the same way as outlined in Sect. 5, and hence, it will not further concern here.

Regarding the latter functionality, the provider of an on-boarded service may at any time request the deprecation, or removal, of the service.[15] Below, we concentrate on deprecation; an entirely symmetrical account applies to removal. We are interested in ensuring that any deprecation actions are performed in a policy-based manner. More specifically, a service *deprecation policy* aims at determining a service *deprecation scheme* by specifying a timescale to deprecation—i.e. the minimum amount of time that must elapse from the receipt of a deprecation request to the actual deprecation action taking place.[16] A BP may incorporate a number of different deprecation policies each specifying a different constraint

[14] It is assumed that an already on-boarded service is updated when its SP submits a fresh SD.

[15] It is assumed that a deprecated service remains in the system but all support to it ceases.

[16] The purpose of such a timescale is twofold: on the one hand, it allows the users of the service under deprecation to switch to one or more services of similar functionality. On the other hand, it allows the providers of any services that depend upon the service under deprecation to resolve these dependencies—e.g. again by replacing the service under deprecation with one or more other services of similar functionality.

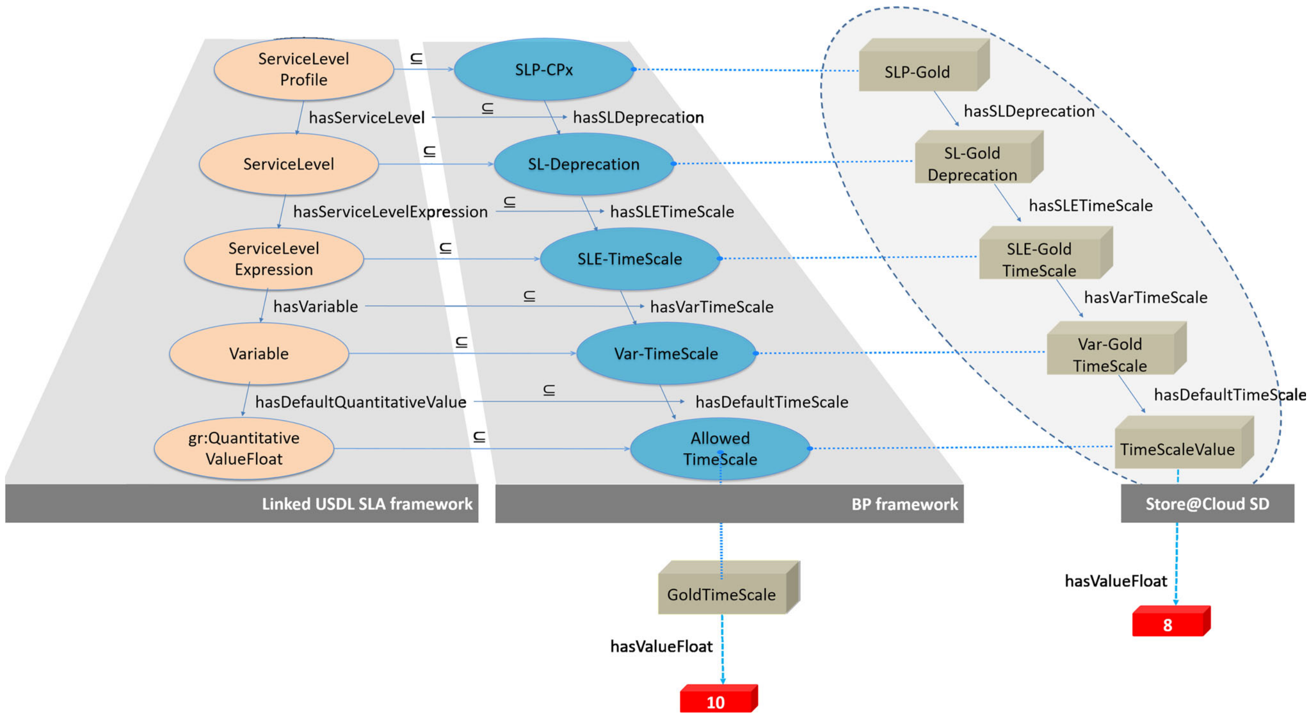

Fig. 4 Deprecation scheme

on the timescale to deprecation. We next briefly elaborate on how such a deprecation policy can be modelled in Linked USDL.

7.1 Modelling service deprecation policies

Linked USDL SLA provides an adequate ontological framework for modelling service deprecation policies. In fact, a service deprecation policy can be considered yet as another SLO, one that corresponds to a *deprecation attribute*. Following the modelling approach outlined in Sect. 3.1.1, such an SLO is ontologically captured as a subclass (say SL-Deprecation) of the class ServiceLevel (see Fig. 4), with particular deprecation SLOs taking the form of instances of SL-Deprecation. Each deprecation SLO is defined in terms of an SLE which constrains the timescale for deprecation. Such a deprecation SLE is modelled as a subclass (SLE-TimeScale) of the class ServiceLevelExpression (see Fig. 4), with particular deprecation SLEs taking the form of instances of this class. A deprecation SLO is associated with its corresponding SLE via the property hasSLETimeScale which is a sub-property of the Linked USDL SLA hasServiceLevelExpression property.

A deprecation SLE articulates its timescale constraint by binding a relevant variable—one which is represented in our model as an instance of the class Var-TimeScale. More specifically, following an approach entirely symmetrical to the one outlined in Sect. 3.1.1, a deprecation SLE is associated through the property hasVarTimeScale (a sub-property of the Linked USDL SLA property hasVariable—see Fig. 4), with an instance of the class Var-TimeScale. This instance is in turn bound, through the property hasDefaultTimeScale (a sub-property of the Linked USDL SLA property hasVariable), with an instance of the class AllowedTimeScale, say the instance GoldTimeScale. The latter instance is associated through the data property hasValueFloat with a particular value that specifies the minimum amount of time that must elapse from the receipt of a deprecation request until the actual deprecation action can take place.

7.2 Enforcing deprecation policies

Each service submitted for on-boarding must specify in its SD a particular deprecation scheme.[17] This is done by incorporating in the SD an appropriate framework of interconnected instances of the aforementioned classes (see Fig. 4). This framework is evaluated for conformance with the corresponding service deprecation policy in the BP in the same manner as outlined in Sect. 5. Upon receipt of a deprecation request for a particular service, the SC^3 proceeds to enforce the deprecation scheme specified in the corresponding SD.

[17] Otherwise, the service cannot be admitted for on-boarding as it cannot conform with the BP.

8 Testing the evaluation mechanisms

A simple *test harness* was created for checking the degree to which the BP and SD evaluation mechanisms operate as expected. The goal of the test harness is to identify cases of invalid BPs or SDs which, nevertheless, manage to pass the evaluation checks. To this end, the test harness automatically creates erroneous versions of BPs (SDs) by introducing syntactic errors into their RDF triples. The BP (SD) evaluation mechanism is then triggered to check these erroneous versions: if one (or more) passes the evaluation, a problem in the evaluation mechanism is inferred.

More specifically, for every RDF triple (s, p, o) in the BP (SD), the test harness introduces an error by substituting s with an erroneous subject s'.[18] It then substitutes (s', p, o) for (s, p, o) in the BP (SD) and runs the evaluation. If, despite the erroneous triple, the BP (SD) passes the evaluation, then this is an indication of a bug in the evaluation mechanism. The same procedure is applied to the rest of the elements of the triple, namely the predicate p and the object o.

The test harness unveiled certain omissions in the checks performed by the evaluation mechanisms. For instance, it was discovered that the SD completenessCheck method failed to check that the properties that interconnect the various instances in an SD are indeed sub-properties of the correct Linked USDL SLA properties. This potentially led to invalid SDs being considered valid. All discovered omissions have been rectified.

9 Related work

To the best of our knowledge, no works other than [5] address the quality assurance dimension of CSB in the context of VEs. [5] recognises the need for frameworks that guide the creation, execution, and management of services in cloud-based VEs; it does not, however, address the quality assurance aspect of such frameworks. The rest of this section outlines works related to service description languages and to ontology-driven policy-based governance and quality control.

9.1 Service description languages

We provide an overview of different strands of service description formalisms. More specifically, we outline approaches that: focus on syntactic service descriptions; consider the underlying semantics of web services; capture business aspects of services.

Syntactic service descriptions aim, primarily, at facilitating the interoperable data exchange between service registries (notably UDDI), service providers, and service consumers. The most prominent example is, perhaps, WSDL [32]. Nevertheless, syntactic service descriptions can only aid manual discovery, selection, and composition of services. In an attempt to automate these processes, a new breed of service description languages was introduced that enable Semantic Web Services [18]. These use ontologies in order to capture the functionality of web services in terms of an underlying, domain specific, vocabulary. The rationale is that since both service descriptions and consumer demands rely on a common semantics, automatic service discovery, and composition is, in principle, feasible. Prominent examples of standardisation efforts in this area include WSMO [30], OWL-S [29], SAWSDL [34], and SA-REST [31].

Whilst focusing on aspects which are important for the automatic composition and invocation of web services, the aforementioned approaches neglect any pertinent business details or, at best, address them as non-functional properties. This renders service descriptions cumbersome for service consumers and third-party intermediaries who are often interested in both business details and technical specifications in order to create added value by deploying, aggregating, customising, and integrating services. A third strand of description languages has therefore emerged, one which focuses on the business aspect of services. A prominent example is the Unified Service Description Language (USDL) [20]. USDL aims at unifying the business, operational, and technical aspects of a service in one coherent description framework. Nevertheless, USDL has received limited adoption due mainly to its complexity and limited support for extensibility. To overcome these limitations, Linked USDL [15] has been proposed. Linked USDL is a remodelled version of USDL which offers the advantages outlined in Sect. 3.

9.2 Ontology-driven policy-based cloud service governance and quality control

Cloud service governance refers to policy-based management of cloud services with emphasis on quality assurance [13]. Current practice [17,37] focuses on the use of registry and repository systems combined with purpose-built software to check the conformance of services with relevant policies [14]. A major weakness in these systems is failure to achieve a separation of concerns between defining policies and evaluating data against these policies [13,14]. This has a number of negative repercussions such as lack of portability, inability to reason about the effectiveness of policies and lack of explicit representation of policy interrelations. Several works have attempted to address these shortcomings [8,12,19,25]. These generally employ bespoke languages, and ontologies, for capturing policies; the policies are then enforced at run-time typically through the use of a reference

[18] s' is derived from s by appending a random character.

monitor. Closer to our approach are the works in [12,19,25] which embrace Semantic Web representations for capturing the knowledge encoded in policies.

In [25], the authors present KAoS—a general-purpose policy management framework which exhibits a three-layered architecture comprising: (i) a human interface layer, which provides a graphical interface for policy specification; (ii) a policy management layer, which uses OWL [33] to encode and manage policy-related knowledge; (iii) a policy monitoring and enforcement layer, which automatically grounds OWL policies to a programmatic format suitable for policy monitoring and enforcement. The latter layer is introduced in order to render the policy monitoring and enforcement processes more efficient. Although a seminal work for its time, the KAoS approach exhibits a number of limitations. Firstly, the programmatic format promoted by KAoS precludes the performance of any updates to the policies *dynamically*, i.e. during system execution, as such updates would naturally require the (updated) policies to be re-compiled to the programmatic format. It also precludes the performance of *semantic inferencing* during policy monitoring and enforcement which invalidates, to a degree, the reason for embracing Semantic Web technologies for capturing the knowledge that resides in policies in the first place.

In [12], the authors propose Rei—a policy specification language expressed in OWL-Lite [33]. Rei allows the declarative representation of a wide range of policies which are purportedly understandable—hence enforceable—by autonomous entities in open, dynamic environments. In Rei, a policy comprises a list of rules that take the form of OWL-Lite properties, as well as a set of contextual attributes (modelled as ontological concepts) that define the underlying policy domain. Rei uses variables in order to express policy rules in which no concrete values are provided for the contextual attributes—e.g. rules of the form 'service s is allowed to access object o only when s is located in the *same area* as another subject s'. Rei resorts to the use of placeholders as in rule-based programming languages for the definition of such variables. This, however, essentially prevents Rei from exploiting the full inferencing potential of OWL as policy rules are expressed in a formalism that is alien to OWL. In contrast, variables could have instead been modelled in terms of OWL's anonymous individuals [33].

In [19], POLICYTAB is proposed for supporting trust negotiation in Semantic Web environments. POLICYTAB advocates an ontology-based approach for describing policies that drive a trust negotiation process aiming at providing controlled access to web resources. Nevertheless, it does not provide any mechanisms for strengthening the resilience of services by checking their conformance against pre-specified policies.

By drawing upon ontological representations of policies, the works presented in [12,19,25] achieve a separation of concerns that disentangles policies from the actual code of the mechanisms through which these policies are enforced. Nevertheless, as already mentioned, these works rely on bespoke, non-standards-based, ontologies which generally lack the expressivity for addressing holistically the business details that characterise web and cloud services. In addition, they hinder multi-party interactions and knowledge sharing and thus reduce the interoperability, reusability and generality of the policies. These limitations have essentially inspired and motivated the work presented in this paper which, by drawing upon the Linked USDL framework, manages to overcome these shortcomings.

10 Conclusions and future work

We have presented the SC^3, a mechanism that strengthens the resilience of services in cloud-based VEs by offering the following capabilities with respect to the Quality Assurance Service Brokerage dimension of CSB:

1. Evaluation of service conformance with pre-specified BPs concerning the business aspects of service delivery.
2. Evaluation of the well-formedness of the BPs themselves.
3. Performance of service lifecycle management in a generic and policy-based manner.

The SC^3 is underpinned by an ontological representation of BPs and SDs, one which is based on Linked USDL and promotes a clear separation of concerns between policy definition and policy enforcement, bringing about the following seminal advantages: (i) it keeps the SC^3 generic and orthogonal to the underlying cloud delivery platform; (ii) it enables generic reasoning about the well-formedness, hence the effectiveness, of the policies; (iii) it enables the identification of inter-policy relations; (iv) it facilitates the overall governance of policies.

The SC^3 has been successfully used, in the frame of EU's Broker@Cloud project [2], for evaluating the quality of CRM services that are on-boarded on an existing commercial cloud application platform—namely the CAS Open [4] platform.

The BP evaluation mechanism offered by the SC^3 (see Sect. 6) is currently only capable of performing *structural checks* that determine whether a BP abides by the ontological framework outlined in Sect. 3. However, it is not capable of assessing whether a BP encompasses all those attributes that are necessary in order to sufficiently specify the business aspects of service delivery that are deemed relevant to a particular cloud platform. For example, it is not able to assess whether a BP correctly constrains each and every attribute of Table 1. Clearly, this is a seminal capability that generally increases our assurance on the quality of the services utilised by cloud-based VEs.

In order to overcome this limitation, as part of future work we intend to extend our ontological representation of BPs by devising a suitable ontological framework, a *higher-level ontology* (HLO), which will enable the generic expression of a set of *constraints* regarding the necessary attributes that any BP must encompass. For instance, going back to the example of Sect. 2, such constraints may insist that any BP must specify exactly one value, or range of values, for each of the attributes of Table 1 and that these values, or value ranges, must fall within the value ranges specified in Table 1; any BP not bearing these characteristics is not considered a correct policy. The HLO will thus constitute a schema, essentially a meta-policy, with which any BP must conform.

The HLO will be formulated in the OWL 2 Web Ontology Language [35] which provides the required expressivity for articulating the aforementioned constraints. For example, it allows the expression of *cardinality constraints* that are necessary in determining all those attributes that are deemed compulsory for a BP. The HLO is anticipated to pave the way for a series of *correctness* checks that are performed automatically by a *policy evaluation* mechanism with reference to the constraints that are encoded in the HLO; this mechanism is intended to replace the current BP evaluation mechanism offered by the SC^3. Moreover, we intend to construct an editor through which a user will be able to prime the HLO with appropriate constraints for a particular domain of application.

Acknowledgements This research was funded by the EU 7th Framework Programme under the Broker@Cloud project (www.broker-cloud.eu), Grant Agreement No. 328392.

References

1. Apache Jena. https://jena.apache.org/
2. Broker@Cloud project: enabling continuous quality assurance and optimisation for cloud brokers. http://www.broker-cloud.eu/
3. Cardoso J, Pedrinaci C, Leidig T, Rupino P, De Leenheer P (2013) Foundations of open semantic service networks. Int J Serv Sci Manag Eng Technol 4(2):1–16. doi:10.4018/jssmet.2013040101
4. CAS CRM. http://www.cas-crm.com/
5. Cretu LG (2012) Cloud-based virtual organization engineering. Informatica Econ 16(1):98–109
6. D30.3 specification of interfaces for enabling brokerage in enterprise cloud service delivery platforms. http://www.broker-cloud.eu/documents (2014)
7. D40.1 methods and mechanisms for cloud service governance and quality control. http://www.broker-cloud.eu/documents (2014). Broker@Cloud Project Deliverable
8. Damianou N, Dulay N, Lupu E, Sloman M (2001) The ponder policy specification language. In: Proceedings of the international workshop on policies for distributed systems and networks, POLICY '01, Springer, London, UK, UK, pp 18–38. http://dl.acm.org/citation.cfm?id=646962.712108
9. Fiksel J (2007) Sustainability and resilience: toward a systems approach. IEEE Eng Manag Rev 35(3):5–5. doi:10.1109/EMR.2007.4296420
10. Foster I, Zhao Y, Raicu I, Lu S (2008) Cloud computing and grid computing 360-degree compared. In: 2008 Grid computing environments workshop, pp 1–10. doi:10.1109/GCE.2008.4738445
11. GoodRelations language reference. http://www.heppnetz.de/ontologies/goodrelations
12. Kagal L, Finin T, Joshi A (2003) A policy language for a pervasive computing environment. In: Proceedings POLICY 2003. IEEE 4th international workshop on policies for distributed systems and networks, pp 63–74. doi:10.1109/POLICY.2003.1206958
13. Kourtesis D, Parakakis I, Simons A (2012) Policy-driven governance in cloud application platforms: an ontology-based approach. In: Proceedings of the 4th international workshop on ontology-driven information systems engineering
14. Kourtesis D, Paraskakis I (2011) A registry and repository system supporting cloud application platform governance. In: Proceedings of the 2011 international conference on service-oriented computing, ICSOC'11, Springer, Berlin, Heidelberg (2012), pp 255–256. doi:10.1007/978-3-642-31875-7_36
15. Linked USDL. http://www.linked-usdl.org/
16. Liu F, Tong J, Mao J, Bohn R, Messina J, Badger L, Leaf D (2011) NIST Cloud computing reference architecture. Technical report NIST
17. Marks EA (2008) Service-oriented architecture governance for the services driven enterprise. Willey, New York
18. McIlraith SG, Son TC, Zeng H (2001) Cloud-based virtual organization engineering. IEEE Intell Syst 16(2):46–53. doi:10.1109/5254.920599
19. Nejdl W, Olmedilla D, Winslett M, Zhang CC (2005) Ontology-based policy specification and management. Springer, Berlin, Heidelberg, pp 290–302. doi:10.1007/1143105320
20. Oberle D, Barros A, Kylau U, H S (2013) A unified description language for human to automated services. Inf Syst 38(1):155–181. doi:10.1016/j.is.2012.06.004
21. Pedrinaci C, Cardoso J, Leidig T Linked USDL (2014) A vocabulary for web-scale service trading. In: Presutti V, d'Amato C, Gandon F, d'Aquin M, Staab S, Tordai A (eds) The semantic web: trends and challenges: 11th international conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014. Proceedings, pp 68–82. Springer International Publishing, Cham. doi:10.1007/978-3-319-07443-6_6
22. RDF—semantic web standards. https://www.w3.org/RDF/
23. SKOS simple knowledge organization system. https://www.w3.org/2004/02/skos/
24. The FOAF project. http://www.foaf-project.org/
25. Uszok A, Bradshaw J, Jeffers R, Johnson M, Tate A, Dalton J, Aitken S (2004) KAoS policy management for semantic web services. IEEE Intell Sys 19(4):32–41
26. Vaquero L, Rodero-Merino L, Caceres J, Lindner M (2008) A break in the clouds: towards a cloud definition. SIGCOMM Comput Commun Rev 39(1):50–55
27. Veloudis S, Friesen A, Paraskakis I, Verginadis Y, Patiniotakis I (2014) Underpinning a cloud brokerage service framework for quality assurance and optimization. In: Proceedings of the 2014 IEEE 6th international conference on cloud computing technology and science, CLOUDCOM '14, IEEE Computer Society, Washington, DC, USA, pp 660–663. doi:10.1109/CloudCom.2014.146
28. Veloudis S, Paraskakis I, Friesen A, Verginadis Y, Patiniotakis I, Rossini A (2014) Continuous quality assurance and optimisation in cloud-based virtual enterprises. In: Camarinha-Matos LM, Afsarmanesh H (eds) Collaborative systems for smart networked environments: 15th IFIP WG 5.5 working conference on virtual enterprises, PRO-VE 2014, Amsterdam, The Netherlands, October 6–8, 2014. Proceedings, Springer Berlin Heidelberg, pp 621–632. doi:10.1007/978-3-662-44745-1-61
29. W3C Member Submission (2004) OWL-S semantic markup for web languages. http://www.w3.org/Submission/OWL-S

30. W3C Member Submission (2005) Web service modelling ontology (WSMO). http://www.w3.org/Submission/WSMO
31. W3C Member Submission (2010) SA-REST: semantic annotations for web resources. http://www.w3.org/Submission/SA-REST
32. W3C recommendation (2001) web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl
33. W3C Recommendation (2004) OWL web ontology language reference. https://www.w3.org/TR/owl-ref/
34. W3C Recommendation (2007) Semantic annotations for WSDL and XML schema. http://www.w3.org/TR/sawsdl
35. W3C Recommendation (2012) OWL 2 web ontology language. https://www.w3.org/TR/owl2-overview/
36. WSO2 carbon 100% open source middleware platform. http://wso2.com/products/carbon/
37. Zhang LJ, Zhou Q (2009) CCOA: cloud computing open architecture. In: Web services, 2009. ICWS 2009. IEEE international conference on, IEEE, pp 607–616