The University Of Sheffield.

Policy-Driven Governance in Cloud Service Ecosystems

By:

Dimitrios Kourtesis

A thesis submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

The University of Sheffield

Faculty of Engineering

Department of Computer Science

28 September 2016

South East European Research Centre

Abstract

Cloud application development platforms facilitate new models of software co-development and forge environments best characterised as cloud service ecosystems. The value of those ecosystems increases exponentially with the addition of more users and third-party services. Growth however breeds complexity and puts reliability at risk, requiring all stakeholders to exercise control over changes in the ecosystem that may affect them. This is a challenge of governance. From the viewpoint of the ecosystem coordinator, governance is about preventing negative ripple effects from new software added to the platform. From the viewpoint of third-party developers and end-users, governance is about ensuring that the cloud services they consume or deliver comply with requirements on a continuous basis.

To facilitate different forms of governance in a cloud service ecosystem we need governance support systems that achieve separation of concerns between the roles of policy provider, governed resource provider and policy evaluator. This calls for better modularisation of the governance support system architecture, decoupling governance policies from policy evaluation engines and governed resources. It also calls for an improved approach to policy engineering with increased automation and efficient exchange of governance policies and related data between ecosystem partners.

The thesis supported by this research is that governance support systems that satisfy such requirements are both feasible and useful to develop through a framework that integrates Semantic Web technologies and Linked Data principles.

The PROBE framework presented in this dissertation comprises four components: (1) a governance ontology serving as shared ecosystem vocabulary for policies and resources; (2) a method for the definition of governance policies; (3) a method for sharing descriptions of governed resources between ecosystem partners; (4) a method for evaluating governance policies against descriptions of governed ecosystem resources. The feasibility and usefulness of PROBE are demonstrated with the help of an industrial case study on cloud service ecosystem governance.

Acknowledgements

My PhD supervisors Iraklis Paraskakis and Anthony J.H. Simons have my gratitude. I am grateful to Iraklis for introducing me to academic research and for giving me the opportunity to work as a research associate at SEERC for six great years. I am grateful to Tony for his wise advice, his intellectually stimulating teaching and his help with keeping me focused on completing this research. I wish to thank both of them for their genuine support at times when full-time work along part-time research study became challenging.

I am thankful to my colleagues at SEERC, to the academics of the Computer Science department at the University's International Faculty in Thessaloniki and to our research partners from companies and academic institutions with whom I had the privilege to work during my PhD research[1].

I also want to thank Andigoni Malousi, my beloved cousin who has been an inspiration for the academic studies I pursued in Computer Science - thank you for helping me to cross the PhD finish line.

The limitless support and loving patience of my wife Aimilia from the beginning of this research to this day is the single thing that kept me going. This work is really her achievement. Aimilia and the two beautiful children we have been blessed with, Yannis and Thanos, have been my motivation.

I am thankful to our son Yannis who is now six years old and planning a future career in science, for being an explosive source of inspiration. I am thankful for his active (daily) encouragement for me to study and complete my writing. At present he finds it extremely cool that dad is pursuing a doctorate degree. I hope he will think the same when he gets to read this dissertation! I am also thankful to our beautiful five-month old son Thanos who has been observing with keen interest and a warm smile but without saying much for the present time —or should I say, not much with clear semantics. Yannis and Thanos will now have more playtime with dad, I promise.

My parents, if it wasn't for you... Mother, I am eternally grateful for your unconditional love and support. Father, I keep your words to my heart and eternally miss you.

[1] This research work was supported by a University of Sheffield Scholarship, with full PhD tuition fees sponsored by the South-East European Research Centre (SEERC). It also benefited from my participation in two research projects sponsored by European Commission research grants: CAST, funded by Eureka Eurostars (E! 4373) and Broker@Cloud, funded by the Seventh Framework Programme (FP7-ICT 318392).

Table of Contents

List of Tables

List of Figures

Το κυβερνάν εστί προβλέπειν.

To govern is to foresee.

(Alcibiades, 450-404 BC)

Chapter 1

Introduction

1 Introduction

1.1 Motivation

The model of cloud computing represents an evolutionary step for human technology analogous to the one that marked the transition from the era of mainframe computing to personal computing [1]. Cloud computing is a transformational force, acting as a catalyst to accelerate developments in a wide range of scientific and industrial fields. One such field is the development of software applications which is now increasingly happening on the cloud.

Platform as a Service (PaaS) is a cloud computing service model that the software industry is adopting at a rapid pace [2]. Cloud application development platforms following the PaaS model have introduced a profoundly different and more efficient way for software creators to develop and deliver web applications.

A key benefit of such platforms for application developers is removing the burden of acquiring, setting up and maintaining their own infrastructure to deliver software over the Internet. Another key benefit is that every cloud application platform includes an array of pre-built application development components or developer services in the form of application programming interfaces (APIs) which offer reusable solutions to recurring problems in application development, so that the time and effort to develop a new piece of software can be greatly reduced [3].

Increasingly, cloud application development platforms follow an open extensible architecture which allows independent third-parties to extend the capabilities of the platform and its array of reusable building blocks with their own add-ons. Third-party extensions add significant value to a platform as they offer more tools and options to the software developers who use the platform to create end-user applications, giving them more solutions for routine tasks or providing them with highly specialised capabilities that would otherwise be challenging or impossible to develop from the ground up.

This model of collaboration between different kinds of software creators which is now made possible by cloud application platforms represents a novel form of software product co-development. The owner and operator of the cloud application platform plays a central coordination role, facilitating and promoting collaboration between all partners. In short, as observed by Hanssen and Dyba [4], software engineering is becoming an open process in a complex distributed environment.

Software platforms which facilitate co-development relationships of this form foster the creation of environments best characterised as software ecosystems [3].

Jansen, Finkelstein and Brinkkemper [5] define a software ecosystem as: "*a set of businesses functioning as a unit and interacting with a shared market for software and*

services, together with the relationships among them. These relationships are frequently underpinned by a common technological platform or market and operate through the exchange of information, resources and artifacts."

The three primary roles in a cloud service ecosystem are: (i) the keystone partner who typically owns the cloud application platform and controls its evolution, (ii) the third-party organisations that use the platform to create user-facing apps or developer extensions, and (iii) the end consumers of apps which are offered by third-parties or by the keystone organisation itself.

The value of an ecosystem for everyone involved increases exponentially with (a) more users and (b) more complementary services built around the platform [6]. But in software, growing in size and diversity is at odds with reliability. The cause of this tension is complexity; a property that emerges as an inevitable side-effect of growth and which is known to be inversely related to software reliability [7],[8].

Managing this complexity is the key to maintaining the reliability of the services that the ecosystem delivers, and ultimately, to maintaining and increasing the ecosystem's value. To manage complexity, ecosystem partners need to be able to exercise control over developments in the ecosystem that may affect them, such as the introduction of a new service, a change to the characteristics of an existing service, or a change to how a set of services is assembled. This is a challenge of governance.

From an organisational viewpoint the challenge of governance lies in establishing an effective and efficient structure for direction-setting and policy-making in the organisation. From a technological viewpoint, the challenge lies in providing effective and efficient tool support to the relevant actors in the organisation such that governance policies can be enforced throughout the lifecycle of the relevant ecosystem resources.

The majority of academics that have so far been writing on subjects related to IT governance have a background in management science or information systems and tend to focus on the first viewpoint, i.e. on how governance decisions can be made in an organisation [9]. In this work we focus on the latter viewpoint, placing emphasis on the policy-driven control mechanisms that are necessary to operationalise those governance decisions. Our focus is on how to create software systems that support policy-driven governance.

The importance of policy-driven governance to control the provision and consumption of cloud services in a software ecosystem is increasingly receiving more attention. From the viewpoint of the ecosystem coordinator, governance is about ensuring that the introduction of new apps and developer services – or the modification of existing ones – will not create a negative impact on the platform's stability and reliability. From the viewpoint of ecosystem participants (third-party developers and end-users) governance is about ensuring that the services they consume or deliver operate as they should, and that they satisfy all relevant requirements on a continuous basis.

The body of literature on the subject of software ecosystem governance has so far focused on governance of the software platform and the ecosystem at large from the viewpoint of the keystone partner. However as cloud service ecosystems mature the role of the keystone partner evolves to include a new type of capability. The platform owner becomes an intermediary to help other ecosystem partners fulfil their individual governance requirements – from the viewpoint of consuming and delivering ecosystem services [10]. Governance becomes an intermediated process involving several distributed actors who assume different types of roles.

This phenomenon is accelerated in cloud service environments of high complexity and is more pronounced in ecosystems involving cloud service brokers who intermediate the consumption and delivery of cloud services [11], [10], [12], [13]. For these reasons, our definition of governance extends beyond the notion of platform governance from the single viewpoint of the ecosystem coordinator, to incorporate the governance viewpoints and requirements of all participants in the ecosystem.

However, examining the state-of-the-art governance technology platforms which are available to the software industry today reveals a gap between the type of requirements these platforms were originally designed to meet and the type of needs emerging to support governance in this new context. The architecture approach adopted by contemporary governance technology platforms embodies certain characteristics which represent critical limitations in relation to governance in cloud service ecosystems.

The root of the problem can be traced in the way these platforms allow governance policies to be defined and evaluated. The policy representation, data extraction logic and policy evaluation logic are typically entangled in the implementation of a single software component which is coded in some imperative (procedural) programming language.

As this dissertation will demonstrate, this represents the strongest form of coupling between three functions that should be kept separate. The consequence is that such systems cannot accommodate usage scenarios where several ecosystem partners need to take part in the governance process. In other words, governance support systems built on these platforms cannot support governance processes where the actor providing a policy may be different from the actor evaluating the policy, or where the latter may be different from the actor providing the data against which a governance policy needs to be checked.

The fact that governance policies, governed resources, and policy evaluation are strongly coupled does not simply make it more difficult for the related stakeholders to manage their governance functions; it also makes it impossible for them to make internal changes and evolve, without creating ripple effects that influence other ecosystem partners.

To enable governance in a continuously evolving cloud service ecosystem we need governance support systems that achieve adequate separation of concerns between the roles of the policy provider, the governed resource data provider and the policy evaluator. Decoupling governance policies, governed resources, and policy evaluation engines allows the associated ecosystem partners to manage their internal governance processes in a more efficient way while they cooperate and coevolve with the rest of the ecosystem.

This calls for better modularisation of the governance support system architecture, allowing decoupling governance policies from policy evaluation engines and governed resources. It also calls for an improved approach to policy engineering that not only enables more automation in policy management but most fundamentally facilitates interoperability and efficient exchange of governance policies and related data between ecosystem partners. The question then arises: How can these goals be achieved? How should governance support system architectures evolve to be able to meet such requirements? What would be a good basis to build on, to achieve this evolution?

As discussed later in this dissertation, this is a problem domain where ontology-driven information systems engineering, ontology-based policy modelling, Semantic Web technologies [14] and Linked Data principles [15] have been successfully applied in the past.

Uschold [16] cites six important benefits which result from the increased level of abstraction and the use of logic in ontology-driven information systems: reduced conceptual gap, increased automation, reduced development times, increased reliability, increased agility and decreased maintenance costs.

On the benefits of applying Semantic Web technologies to policy engineering Tonti et al. [17] highlight reduced human error, simplified policy analysis, reduced policy conflicts, and increased interoperability, while Uszok et al. [18] emphasise reusability, extensibility, verifiability, safety, and automated reasoning.

Linked Data principles on the other hand provide the key benefit of efficient integrated access to data from distributed and heterogeneous data sources [15], raising data interoperability to an entirely new level. As noted by Heath and Bizer [19], the premise underlying Linked Data is that "*just as the World Wide Web has revolutionised the way we connect and consume documents, so can it revolutionise the way we discover, access, integrate and use data*" [19].

The view that this research puts forward is that the basis for achieving an evolutionary step in the design of governance support systems for software ecosystems can be provided by a new approach to the definition and enforcement of governance policies in which Semantic Web technologies, Linked Data principles, and knowledge representation and reasoning will have a central role.

1.2 Research aims and objectives

The aim of the research presented in this dissertation has been to investigate the *feasibility* and *usefulness* of a new software framework which integrates Semantic Web technologies and Linked Data principles to meet the advanced governance requirements posed by cloud service ecosystems. In the chapters to follow we refer to this framework as PROBE (policy-driven governance in cloud service ecosystems).

The intermediate objectives to achieve the aim of the research can be summarised as follows:

- Analyse an industrial cloud service ecosystem case study and other relevant examples, survey the literature in software ecosystem governance and develop a working definition for the concept of governance in cloud service ecosystems.
- Develop a model of design requirements and software architecture quality attributes for governance support systems which reflects the needs of governance processes in cloud service ecosystems.
- Survey state of the art service governance technology and known applications in cloud service environments to identify limitations and dimensions of required enhancements with respect to the previously derived requirements model.
- Define a conceptual framework to help software engineers develop governance support systems capable of meeting the specific requirements of cloud service ecosystems, by integrating Semantic Web and Linked Data technologies.
- Survey policy ontologies and experiment with alternative ontology modelling approaches based on an industrial cloud service ecosystem case study, to develop an ontology serving as shared governance policy vocabulary.
- Experiment with alternative ontology-based policy modelling approaches, develop a method for policy definition and policy checking utilising the previous case study, and implement a prototype policy evaluation engine.
- Develop guidelines for producing or automatically generating descriptions of governed ecosystem resources which are based on the same ontology vocabulary and can be automatically verified against ontology-based policies.
- Validate the completeness of the developed methods and the implemented prototype by applying them on the governance policies derived from the industrial cloud service ecosystem case study.
- Assess the relative advantages and disadvantages of the new framework by comparing it to a solution achievable with a state-of-the-art governance platform, using the same industrial cloud service ecosystem case study.

1.3 Research results

In fulfilment of the objectives listed above this research work delivered the following results:

1. Framing requirements thinking for governance support systems used in cloud service ecosystems in terms of the individual viewpoints of the different types of roles who participate in the governance process. This includes conceptualising the need to decouple governance policies, governed resource data and policy evaluation engines, so as to facilitate separation of concerns between the different governance process roles and to allow ecosystem partners to cooperate and coevolve in an agile manner.
2. Defining PROBE as a new conceptual framework integrating Semantic Web technologies and Linked Data principles to help develop governance support systems that meet the advanced requirements of cloud service ecosystems.
3. Demonstrating the feasibility of the PROBE framework by developing a concrete instantiation of its components using an industrial cloud service ecosystem case study as source of requirements and use cases. This contribution is delivered through the following individual results:
    a. Developing a governance ontology serving as shared conceptual model between ecosystem partners to define policies and to describe governed resources in a way that is abstract, amenable to automated analysis and interoperable.
    b. Developing a method for ontology-based definition of governance policies including policy modelling patterns for different types of governance policies and different forms of policy expression.
    c. Developing guidelines for the creation of structured, interoperable and highly reusable ontology-based descriptions of governed resources, and sharing them among ecosystem partners with Web standards.
    d. Developing a prototype of a generic logic-based policy evaluation engine which allows distributed policy evaluators to check policies against governed resource data without requiring any customisation.
4. Demonstrating the usefulness of the PROBE framework through a comparison to the type of solutions afforded by state-of-the-art governance technology platforms and a side-by-side assessment using the same cloud service ecosystem case study.

1.4 Dissertation outline

Chapter 2 – 'Cloud service ecosystems and governance support systems' presents the background to this research work. It provides an introduction to the paradigm of cloud computing and the different cloud service models available today with a focus on cloud application development platforms. A discussion on the different types of software co-development models afforded by cloud application platforms leads to introducing cloud service ecosystems as a special class of software ecosystems. It also leads to discussing how the complexity of software ecosystems gives rise to governance as a critical requirement. The theme of governance is introduced through alternative viewpoints, followed by our own definition of governance in cloud service ecosystems and a survey of the most relevant research.

Chapter 3 – 'Governance in cloud service ecosystems: key requirements' lays the groundwork for requirements thinking on the challenge of governance in cloud service ecosystems. It provides an analysis of the distinct roles and individual concerns for different ecosystem actors who may be stakeholders in governance processes. To help outline how these roles function and to illustrate their different types of concerns in full variance, the chapter opens with five exemplifying scenarios. An analysis follows of the implications emerging for the design of governance support systems for cloud service ecosystems, highlighting the need for separation of concerns between the roles and interfaces of the policy provider, the governed resource data provider and the policy evaluator. The chapter concludes with summarising the design principles and architecture quality attributes for governance support systems based on the requirements of different governance process roles.

Chapter 4 – 'A new foundation for governance support systems' introduces the thesis supported by the research work and this dissertation. The chapter opens with an analysis of how policy-based governance is facilitated by contemporary governance support systems, based on a study of two commercial governance technology platforms which are also open-source. Their limitations with respect to the requirements analysed in the previous chapter are highlighted and dimensions of required changes are identified. The chapter continues with introducing logic-based knowledge representation and reasoning, ontology modelling, Semantic Web technologies and ontology-driven information systems engineering as the foundation for a new approach to the development of governance support systems. This leads to presenting the thesis statement: Governance support systems that satisfy the evolved governance requirements of cloud service ecosystems are both feasible and useful to develop with an architecture framework that integrates Semantic Web technologies and Linked Data principles. The chapter concludes with presenting the conceptual architecture framework of PROBE (policy-driven governance in cloud service ecosystems).

Chapter 5 – 'Defining governance policies' presents the instantiation of the first two components of the PROBE framework as introduced in chapter 4. The first component

is a governance ontology encoded in OWL-DL [20] which serves as shared vocabulary between ecosystem partners for policy definition and data description. The second component is a method for the definition of governance policies by different ecosystem partners, based on the shared governance ontology. Process and resource governance policies are formulated in either positive or negative form and encoded as OWL class axioms and SWRL [21] rules. Policy examples from project CAST [3], [22], an industrial case study on cloud service ecosystem governance, are utilised as use cases. The chapter concludes with relevant work on ontology-based policy representation and related semantic technologies.

Chapter 6 – 'Describing governed resources' discusses the instantiation of the third component of the PROBE framework as introduced in chapter 4: methods to create RDF [20] descriptions of governed resources and to share them between ecosystem partners. An approach is described that combines transformation mappings of native data representations against the governance ontology, dynamic on-demand generation of RDF triples and SPARQL-based access [23]. Governed resource examples from project CAST are again utilised as use cases. The chapter concludes with design guidelines for setting up a Linked Data provision and sharing architecture and a review of related work on enabling technologies.

Chapter 7 – 'Evaluating governance policies' presents the instantiation of the final remaining PROBE framework component: a mechanism to evaluate governed resource descriptions against governance policies when both have been defined and described on the basis of a shared governance ontology. The background to OWL-based data validation and alternative computation approaches to the problem of policy evaluation are discussed. A method and prototype implementation is described which allows overcoming the challenges presented by certain characteristics in the default language semantics of OWL to enable automated policy evaluation reasoning. Examples from project CAST are again utilised as use cases. The chapter concludes with relevant work on formulation of constraints and automatic validation of data.

Chapter 8 – 'Comparative case study' presents a comparative assessment of alternative governance support system architectures which demonstrates the advantages of the PROBE framework over the solutions afforded by state-of-the-art governance support systems. The first design approach described is the one that project CAST originally adopted to develop the governance support system for the CAST cloud application platform. The second approach is the one proposed by the PROBE framework as described in chapters 5, 6 and 7. The two approaches are compared based on the same case study: the governance policies of project CAST. They are then examined from the perspective of the different roles involved in the ecosystem governance process and change-scenarios are used to evaluate how each approach supports evolvability and manageability of the governance process. This assessment demonstrates the usefulness of implementing the PROBE framework in a complex industrial setting and highlights the framework's strengths.

Chapter 9 – ‘Conclusions’ is the final chapter which brings the dissertation to a close by returning to the aims and objectives of this research work. The chapter provides an overview of the research carried out and the key contributions achieved. It discusses significance of the results and closes with a description of limitations and directions for further research.

Chapter 2

Cloud service ecosystems and governance support systems

2 Cloud service ecosystems and governance support systems

2.1 Introduction

In this chapter we present the background and wider context to the research work in this dissertation. We start with an introduction to cloud computing and cloud service models and focus on the type of cloud application development platforms which are often referred to as Platform as a Service offerings.

We then discuss the model of software co-development in the context of cloud application platforms as a significant transformational force in the cloud services market. Following, we provide background to the concept of software ecosystems and discuss cloud service ecosystems as a manifestation of software ecosystems in the context of cloud services.

The next section discusses the challenge of governance in cloud service ecosystems. We start with disambiguating governance as a term, discussing different definitions and viewpoints. We then survey relevant research under the theme of software ecosystem governance and develop our own working definition of cloud service ecosystem governance.

Finally, we discuss governance support systems. We discuss examples of governance control mechanisms from different software ecosystems. We explain how governance support systems have historically evolved and provide an overview of capabilities and characteristics of commercial solutions for governance support in service-oriented architectures. Finally, we discuss the suitability of such systems for applications in governance of cloud service ecosystems.

2.2 Cloud services

2.2.1 The paradigm of cloud computing

The term cloud computing refers to the concept of remote provisioning of pooled computing resources which are made available over a network, in a dynamic, on-demand and scalable fashion, and whose consumption is metered to enable usage-based billing. This notion is a departure from the established paradigm of computing where consumers of computing resources, ranging from single users to entire organisations, are required to buy and maintain their own hardware and software in order to have computing capabilities at their disposal.

Cloud computing can be seen as a move towards fulfilling the vision of utility computing, i.e. a vision of a future where computing resources will be provisioned and consumed as public utilities like electricity or telephony [24]. This vision is certainly not new, as it can be traced back to the 1960s when John McCarthy [25] and Douglas Parkhill [26] pioneered the idea. What makes it seem possible to achieve at this point in time is the level of maturity that Internet and Web technology have reached over the past two decades. The term cloud computing itself, was inspired from the cloud figure that is frequently used to represent the Internet in contemporary telecommunication and software system diagrams [27].

The first spoken reference to the term "cloud computing" in the modern sense was by Eric Schmidt, Google CEO in August 2006: "*It starts with the premise that the data services and architecture should be on servers. We call it cloud computing—they should be in a 'cloud' somewhere.*" [28]. A few weeks later Amazon announced their Elastic Compute Cloud (EC2) offering and the term found widespread adoption.

Adoption of cloud computing has been advancing rapidly. It is universally recognised as a model with tremendous potential for technological and business innovation and has become subject of intense debates with proponents and critics. Despite the hype that surrounds the topic for several years now [29],[30] cloud computing is not an ephemeral concept. The arrival of cloud computing signals a new era in computing and represents an evolutionary step analogous to the one that marked the transition from the era of mainframe computing to personal computing [1].

2.2.2 Cloud computing service models

A number of definitions for cloud computing have been proposed, with each placing emphasis on different aspects of the concept [29],[31]. Despite the differences, there appears to be general consensus regarding the range of cloud computing service models. Most commonly, cloud computing services are classified under the models of Software as a Service (SaaS), Infrastructure as a Service (IaaS), or Platform as a Service (PaaS) [32] although boundaries between the last two models are increasingly blurring in state-of-the-art cloud service offerings.

Software as a Service (SaaS)

> Software as a Service (SaaS) refers to the concept of making software applications accessible in an on-demand fashion, typically through a thin client running inside a Web browser, and under a pay-as-you-go subscription fee (e.g. paid on a monthly or yearly basis). This model is a departure from the established practice of making software applications available as-a-product, i.e. in a form which requires distribution and on-premise installation and maintenance by the user. At the same time, it represents an evolved version of the ASP (Application Service Provider) model for Internet-based application delivery which was popularised during the 1990s. In contrast to the ASP model,

which involved maintaining a separate copy and running instance of an application for each of the provider's client organisations, the SaaS model presupposes a single-instance/multi-tenant application architecture, which allows serving multiple client organisations with a shared application codebase and shared application runtime environment.

Infrastructure as a Service (IaaS)

In an analogy to the SaaS model, Infrastructure as a Service (IaaS) refers to the notion of providing on-demand access to computing infrastructure, over the Internet or some private network, while metering the usage of computing resources and charging the corresponding service fees. The infrastructure being provisioned can be raw computing infrastructure (data storage, processing and networking capacity), server software infrastructure (operating systems, database management systems and Web application servers), or a combination of both.

Platform as a Service (PaaS)

Platform as a Service (PaaS) refers to the concept of combining a particular computing infrastructure and server software stack which can be accessed over some network, with a stack of software tools and services that enable software developers to create software applications and deploy them on the platform.

The deployed applications can be subsequently consumed by end-users in an on-demand fashion. The platform assumes the responsibility to monitor the usage of every application and to allocate infrastructure resources as appropriate to meet usage demand. Often, the platform owner is compensated by the application developer for the amount of infrastructure resources that an application consumes for as long it is being used. However in the case of commercial applications the platform's compensation may also be in the form of revenue sharing from application subscriptions.

Offerings following the PaaS model are primarily focusing on two target groups. The first group is Independent Software Vendors (ISVs), i.e. companies who are looking to create and market on-demand Web applications addressable to large numbers of potential customers in niche domains. The second group is internal IT teams, looking to create solutions for specific needs of users within their own organisations [33].

For both groups, using a PaaS offering generally shifts a significant share of the concerns associated with developing, maintaining and provisioning on-demand software to the platform provider's end. This allows developers to focus on the functionality of their applications, rather than the functionality of the infrastructure that would be required to make those applications available.

The separation of responsibilities between cloud service provider and cloud service subscribers in different cloud service models is illustrated in the figure below by Yung Chou [34].

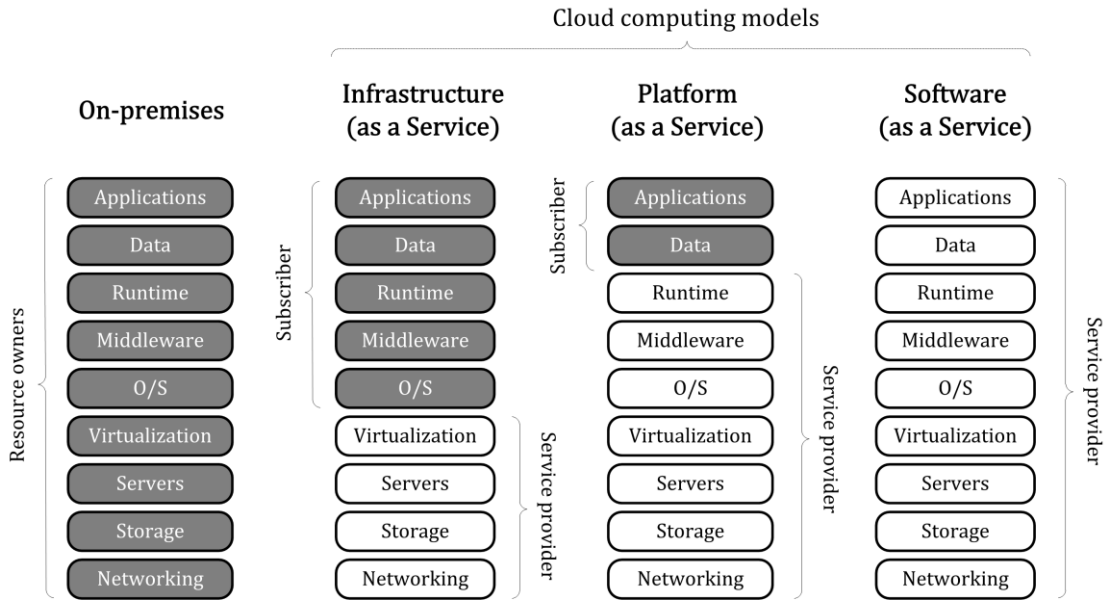

Figure 1. Separation of responsibilities in cloud computing service models. Adapted from [34].

In the following sections we will focus on the co-development possibilities and ecosystem creation potential presented by cloud application development platforms which fall under the definition of the PaaS service model. We will be using the terms 'cloud application development platform' and 'cloud application platform' interchangeably.

2.2.3 Cloud application platforms

The benefits that application developers can gain by adopting a PaaS offering depend on the characteristics of the particular platform. In broad terms, cloud application development platforms which are delivered as PaaS offerings could be classified as platforms with domain-agnostic or domain-specific orientation.

Domain-agnostic orientation

> Cloud application platforms that provide a computing infrastructure and server software stack for the development and delivery of Web-based applications using a particular programming language or framework (e.g. Java, Python, .NET, Ruby), independently of any specific application domain. Examples include Google App Engine[2], Microsoft Azure[3] and Heroku[4].

[2] https://cloud.google.com/appengine/
[3] https://azure.microsoft.com/

Domain-specific orientation

> Cloud application platforms that provide analogous facilities for the development and execution of applications as above, but additionally include components and application programming interfaces (APIs) specialised to a specific domain. Examples of such platforms include SAP YaaS [5] for ecommerce, Force.com [6] for customer relationship management, Intuit Developer[7] for accounting and Zoho Creator[8] for situational applications.

A fundamental benefit that both types of platforms bring to application developers is removing the burden of acquiring, setting up, and maintaining their own infrastructure for provisioning software over the Internet. The cost of hardware, software and networking bandwidth, but also the effort of performing installations, regular backups, emergency upgrades, or any other form of maintenance to the infrastructure, are concerns of the platform provider.

Another fundamental benefit is that every platform includes an array of pre-built components or services which offer reusable solutions to recurring problems in Web applications engineering, so that the time and effort to develop a new application can be reduced.

Increasingly, cloud application platforms follow an architecture which allows third-parties to enrich the set of reusable development building blocks offered by the platform provider with their own add-ons. In this way, cloud application developers can be relieved from many tasks which are either routine in Web application development (such as implementing a mechanism for database access or user authentication), or are highly specialised and particularly challenging (such as implementing a mechanism for load balancing). Each cloud services platform provides an array of built-in or third-party components that address aspects like these; developers need only be concerned with constructing their applications such that they are compatible with the platform, and able to leverage those mechanisms. Examples include Heroku Add-ons listed on Heroku Elements Marketplace[9] or YaaS Packages listed on YaaS Market[10].

Lastly, another important benefit of cloud application platforms specifically for developers of commercial applications (ISVs) is that they make it easier for applications to be marketed and distributed to potential users. This is achieved by means of *app stores* or *app marketplaces* [35]; a concept recently popularised by mobile apps. Cloud app stores/marketplaces are operated by the same company that offers the cloud application platform and provide listings and descriptions of

---

[4] https://www.heroku.com/
[5] https://www.yaas.io/
[6] https://www.salesforce.com/products/platform/products/force/
[7] https://developer.intuit.com/
[8] https://www.zoho.com/creator/
[9] https://elements.heroku.com/
[10] https://market.yaas.io

third-party applications available for end-users to purchase. Microsoft Azure Marketplace[11], Salesforce AppExchange[12], Zoho Marketplace[13], Intuit QuickBooks App Store[14], and Google Apps Marketplace[15] are examples from the platforms already mentioned.

2.3 Software ecosystems in the cloud

2.3.1 Software co-development

The above described new model of collaboration between creators of software that is made possible by cloud application platforms represents a novel form of software product co-development, which has been accelerated by all the recent advancements in cloud computing [3].

Collaborative product development [36],[37] has been growing in importance over the past decades in various industry areas, and software co-development can be seen as a manifestation of this phenomenon in the field of software [3]. For many years, software companies have been practising the development of commercial software products in relative isolation from others in their industry [38]. At some point though, software vendors started realising the benefits of partnerships beyond the obvious model of software distribution, and started opening their products to co-development [39]. Initially it was large-scale software products, notably operating systems, that started to transform from single-vendor projects into joint platform efforts [40],[41] but co-development models quickly found applications in software of varying size and complexity. The previously "fixed" supply chain model of collaboration in the software industry has started giving way to new partnership approaches where large numbers of partners can add value to a co-development platform [42]. There can be advantages for everyone involved: reduced costs, improved focus, reduced complexity, quicker time-to-market and consequently improved economics [38].

In this new context of co-development the software platform owner performs a central coordination role to facilitate and promote collaboration. In some cases the software platform is open for all interested parties to contribute with their resources without any control by the platform owner. In other cases the platform is effectively closed, with the platform owner being in control of access levels and vetting the contributions made by third-parties.

Software co-development models can manifest in different forms depending on the architecture of the software platform.

[11] https://azure.microsoft.com/marketplace/
[12] https://appexchange.salesforce.com/
[13] https://marketplace.zoho.com
[14] https://apps.intuit.com/
[15] https://apps.google.com/marketplace

Basic co-development models

The most elementary form of co-development is when software platforms are extended by new user-facing applications created by third-parties. Developers of such extensions are effectively contributors who add value to the platform by extending its capabilities. Mobile apps for Apple iOS and Android, or desktop browser extensions for Mozilla Firefox and Google Chrome are some obvious examples. In the context of cloud services, Google Apps for Work and Force.com are examples of platforms which encourage third-parties to extend their functionality with new user-facing apps.

Advanced co-development models

Software platforms which can be extended in more sophisticated ways facilitate more advanced forms of co-development. In addition to extending the platform's capabilities via user-facing apps, some software platforms allow adding to the platform's capabilities via reusable software building blocks or developer services. These are accessible through APIs that other developers can subsequently use in creating their own user-facing apps. In this setting contributors to the co-development platform can build on other contributors' work. One relevant example from the cloud application platforms already mentioned is the YaaS (Hybris as a Service) ecommerce platform by SAP.

Jansen and van Capelleveen [43] refer to these two different types of co-development models as "first-generation" and "second-generation" extension models. The difference between them is that in the second-generation extension model an extension can be "*the consumer as well as the provider of resources and services*" [43]. Second generation platform extension architectures allow extensions to interact and have dependency relations between them.

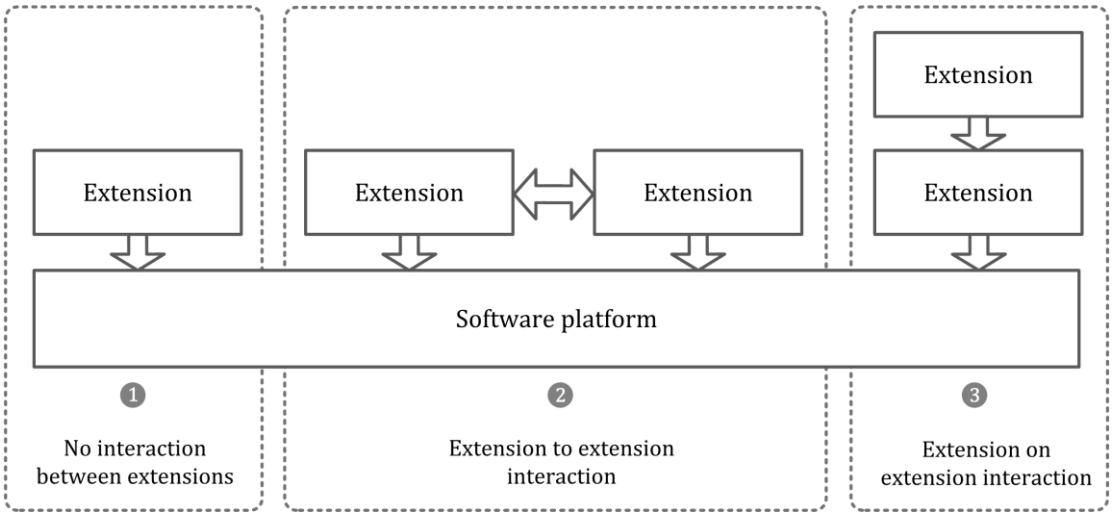

Figure 2. Typology of platform extension models. Adapted from [43].

Advanced models of platform extensibility and software co-development allow relationships to be formed not only between the platform owner and individual contributors, but most importantly, between contributors themselves. This gives rise to a richer model of many-to-many co-development relationships, as opposed to traditional one-to-one co-development collaboration [3]. In short, as observed by Hanssen and Dyba [4], software engineering is becoming an open process in a complex distributed environment.

2.3.2 Software ecosystems

Software platforms which facilitate co-development relationships between different partners in the industry foster the creation of environments best characterised as *software ecosystems*.

The term was first introduced by Messerschmitt and Szyperski in 2003, who defined a software ecosystem as "a collection of software products that have some given degree of symbiotic relationships" [42]. Over the decade that followed the concept of software ecosystems became established as a new paradigm in software engineering, proposing "*participative engineering across independent development organisations centred on a common technology*" [44].

The year of 2009 was a turning point for software ecosystems research as the field started showing signs of consolidation and different researchers who were independently working on the subject published alternative definitions of the term. Kittlaus and Clough defined a software ecosystem as "*an informal network of (legally independent) units that have a positive influence on the economic success of a software product and benefit from it*" [45]. Bosch and Bosch-Sijtsema defined a software ecosystem as consisting of "*a software platform, a set of internal and external developers and a community of domain experts in service to a community of users that compose relevant solution elements to satisfy their needs*" [46].

The definition which is most widely used in related literature [47] is the one by Jansen, Finkelstein and Brinkkemper [5] who define software ecosystem as:

> *"a set of businesses functioning as a unit and interacting with a shared market for software and services, together with the relationships among them. These relationships are frequently underpinned by a common technological platform or market and operate through the exchange of information, resources and artifacts."*

As observed by Jansen and Cusumano [48] software ecosystems are a relatively new concept but in essence they represent a subclass of business ecosystems as introduced in the 1990s by James F. Moore. In his book titled "The Death of Competition" [49] Moore defined business ecosystems as:

> *"An economic community supported by a foundation of interacting organizations and individuals: the organisms of the business world. This economic community produces goods and services of value to customers, who are themselves members of the ecosystem. The member organizations also include suppliers, lead producers, competitors, and other stakeholders. Over time, they coevolve their capabilities and roles, and tend to align themselves with the directions set by one or more central companies."*

In a survey of the literature that contributes to the development of software ecosystems theory, Hanssen and Dyba [4] identify three primary roles in a software ecosystem.

- *Keystone:* The organisation which acts as the "keystone" and coordinates the development of the common technological platform.
- *Third-parties:* The third-party organisations that use the platform to develop solutions or services.
- *Users:* The end-users of those solutions and services.

Software ecosystem researchers use different terms for these roles. Rickmann et al. [50] use the terms "platform sponsor", "complementors" and "customers" respectively, to describe the same notions as above. For the platform sponsor's role Jansen and Cusumano use the term "ecosystem coordinator" [48], while van Angeren et al. refer to the same function as "ecosystem orchestrator" [51]. Iansiti and Levien [52] refer to complementors as "niche players" that deliver products for special niche markets while Eisenmann et al. [53] call them "supply side platform users".

In addition to these primary roles, various other entities may participate in the ecosystem in different capacities, such as standardisation and certification organisations, distributors, resellers, and several others [4].

One of the defining characteristics of software ecosystems is the presence of indirect network effects. As defined by Scott Shane [54] "*In markets with indirect network effects, the value of any component does not depend directly on the number of other users of that component (hence the terminology), but rather on the availability of complementary and compatible components. For example, a PC is more valuable as the set of available software for that PC grows.*"

2.3.3 Cloud service ecosystems

By extension, we can recognise the same characteristics of business ecosystems and software ecosystems in environments where the "common technological platform", according to Jansen's definition [5] is a cloud application platform. In the rest of this dissertation we will be referring to this special class of software ecosystems as *cloud service ecosystems*.

Unlike an ecosystem where the common technological platform is restricted to the technology with which ecosystem partners develop software, a cloud application platform also provides the technology through which this software is delivered to end-users.

Referring back to the definition of ecosystem roles by [4], the primary roles in a cloud service ecosystem are distributed as follows:

- *Keystone:* The organisation that owns the cloud application platform and controls its evolution, such as SAP in the case of YaaS or Salesforce for Force.com has the role of the "keystone" [55] or "ecosystem coordinator" [48].
- *Third-parties:* The organisations that develop user-facing cloud apps or developer services which integrate with the platform are the third-party organisations.
- *Users:* The consumers of the services which are created and offered by third-parties or by the keystone organisation are the end-users.

In sections 2.2.3 and 2.3.1 above we mentioned some examples of cloud application platforms which span domain-agnostic to domain-specific orientation and facilitate basic to advanced co-development models. In the following paragraphs we will expand on some of those examples of platforms.

Case 1: Microsoft Azure

Microsoft Azure[16] is a domain-agnostic cloud application platform providing an advanced co-development model. It is a cloud application platform for developing software applications using a broad selection of programming languages, frameworks and tools, including JavaScript, Python, .NET, PHP, Java and Node.js. An integral part of the platform is the Microsoft Azure Marketplace[17]. Azure offers many different ways for third-parties to integrate and add value to the platform. Among other options, Azure App Service allows third-party ISVs to create full-blown SaaS solutions or develop API-accessible apps for other developers to reuse. Publishing apps and developer services in the Azure marketplace allows ISVs to reach a global market of users.

Case 2: Heroku

Heroku[18] is another domain-agnostic cloud application platform providing an advanced co-development model. Heroku reached a critical mass of users as the dominant cloud platform for developing applications in Ruby. Except for Ruby, Heroku developers can deploy code in Node.js, Java, PHP, Python, Go, Scala and Clojure. Third-parties can use Heroku to build stand-alone SaaS solutions

[16] https://azure.microsoft.com/
[17] https://azure.microsoft.com/marketplace/
[18] https://www.heroku.com/

or use a self-service portal and development kit to offer new developer services as add-ons to the Heroku platform. The third-party add-ons can be either developed and hosted on the third-party's infrastructure or developed and hosted on Heroku itself. At present, there are reportedly more than 150 developer add-ons on Heroku Elements Marketplace[19].

Case 3: Force.com

Force.com[20] is a domain-specific cloud application platform providing a basic co-development model. It is offered by Salesforce.com – presently a dominant SaaS vendor in customer relationship management (CRM) software. Force.com allows third-parties to develop custom web apps that can be deployed on its platform. The apps can be used either independently as stand-alone business applications or as extensions that integrate with –and add capabilities to– Salesforce's range of products. Developers have the option to publish the apps they create to Salesforce AppExchange marketplace[21] allowing end-users to find and buy them. AppExchange is currently said to include over 3,000 pre-integrated apps built to extend the capabilities of Salesforce products.

Case 4: SAP YaaS

YaaS[22] (also known as Hybris as a Service) is a domain-specific cloud application platform providing an advanced co-development model. It is offered by SAP who specialise in enterprise resource planning (ERP) systems and is based on the technology that SAP acquired from its acquisition of Hybris ecommerce platform in 2013. YaaS is focused on enabling development of cross-channel commerce applications with an "API-first" approach. It allows businesses to develop full-scale ecommerce sites and mobile ecommerce apps which are deployed and run on SAP's cloud infrastructure. It also allows developers to create microservices with RESTful APIs [56] that bring new ecommerce capabilities to the Hybris platform (such as content personalisation, or advanced analytics) that others can reuse in creating Hybris-based ecommerce solutions. Third-party apps and developer APIs are listed as packages on YaaS Market[23].

Gawer and Cusumano [6] make the observation that the value of a software platform increases exponentially with (a) more users and (b) more complementary products and services built around the platform. Indeed, the value of a cloud service ecosystem for its members is determined by the number of third-party partners and users it attracts, the diversity of the services it makes available, and the reliability of these services. When a

[19] https://elements.heroku.com/
[20] https://www.salesforce.com/products/platform/products/force/
[21] https://appexchange.salesforce.com/
[22] https://www.yaas.io/
[23] https://market.yaas.io

cloud service ecosystem increases in size, diversity and reliability, its value also increases for all stakeholders involved.

But in software, size and diversity are at odds with reliability. The cause of this tension is complexity; a property that emerges as an inevitable side-effect of growth and is known to be inversely related to software reliability [7],[8]. Managing the deleterious effects of complexity in a cloud service ecosystem is therefore key to maintaining the reliability of the services that the ecosystem delivers, and ultimately, to maintaining and increasing the ecosystem's value.

To manage complexity, ecosystem partners need to be able to exercise control over developments in the ecosystem that may affect them, such as the introduction of a new service, a change to the characteristics of an existing service, or a change to how a set of services is assembled. This can be understood as a challenge of governance, and it can mean very different things to different stakeholders in a cloud service ecosystem.

2.4 The challenge of governance

2.4.1 Definitions of governance

Governance is a broad term which is used in diverse contexts. The English verb *govern* derives from the Greek verb *kybernan* (κυβερνάν), meaning "to direct", "to steer" [57]. Political scientist Mark Bevir defines governance as "*all of processes of governing, whether undertaken by a government, market or network, whether over a family, tribe, formal or informal organisation or territory and whether through the laws, norms, power or language*" [58].

In IBM's view [59], enterprise governance involves two components:

> "*Establishing chains of responsibility, authority and communication to empower people with decision rights.*
>
> *Establishing measurement, policy and control mechanisms to enable people to carry out their roles and responsibilities."*

IT governance is defined by IBM as a facet of enterprise governance focusing on "*an organization's information technology processes and the way those processes support the goals of the business*" [59]. The IT Governance Institute (ITGI) defines IT governance in a similar way: "*leadership and organisational structures and processes that ensure that the organisation's IT sustains and extends the organisation's strategies and objectives.*" [60].

For enterprises whose IT infrastructure adopts a service-oriented architecture (SOA) [61] IBM defines SOA governance as "*an extension of IT governance specifically*

focused on the lifecycle of services, metadata and composite applications in an organization's service-oriented architecture." [59].

With the rapid adoption of cloud computing the services that a modern organisation relies on are increasingly "cloud-delivered" rather than "on-premise" [62], which brings forth the notion of *cloud service governance* as yet another extension of IT governance. Lithicum defines cloud service governance as "*the ability to define, track, and monitor service execution on any number of on-premise and cloud-based platforms*." [62].

In all its different expressions, governance is fundamentally a cross-disciplinary subject that can be viewed from both an organisational and technological perspective. For instance, this is made clear in IBM's definition [59], which encompasses the component of organisational structure and decision-making, as well as the component of policy and control mechanisms.

From an organisational viewpoint the challenge of governance lies in establishing an effective and efficient structure for direction-setting in the organisation. From a technological viewpoint, the challenge lies in providing effective and efficient tool support to the relevant actors in the organisation such that governance goals, expressed as policies, can be enforced throughout the lifecycle of all services.

As noted by Papageorgiou et al. [63] it is useful to distinguish between the overall governance procedure and the corresponding technical support mechanisms, i.e., the system that facilitates, enables and/or automates governance aspects.

Most academics who have been writing on subjects related to governance have a background in management science or information systems and tend to focus on the first viewpoint, i.e. how governance decisions can be made [9]. In this work we focus on the latter viewpoint, placing emphasis on the policy-driven control mechanisms that are necessary to effect those governance decisions. The focus of this research is on software systems supporting policy-driven governance.

2.4.2 Governance in cloud service ecosystems

Cloud service ecosystems are complex environments composed of many participants, each of which may have their own view of the ecosystem, their own objectives from participating and their own governance needs. Governance can mean different things to different types of stakeholders.

We define governance in cloud service ecosystems as the process and the supporting systems for defining and enforcing policies to control the creation, provision and consumption of cloud services by different ecosystem partners.

Our definition encompasses two forms of policy-driven governance.

- *Process governance:* defining and enforcing policies to ensure that the cloud services which are provided and used by the ecosystem are created and modified following an explicit process and lifecycle rules.
- *Resource governance:* defining and enforcing policies to ensure that the artefacts associated with the cloud services which are provided and used by the ecosystem conform to explicit content and structure rules.

When discussing governance in the context of a software ecosystem the view typically taken in literature is that we talk about how the ecosystem coordinator is governing the common software platform, or the activities of ecosystem partners. Uludag et al. [64] refer to these two types of governance as "platform governance" and "ecosystem governance", respectively. However, governance *in* a cloud service ecosystem entails more than the governance *of* the ecosystem. The latter is only one viewpoint – albeit a very important one as it reflects the immediate concerns of the platform owner.

Viewpoint of ecosystem coordinator

> By design, a cloud application platform is an open environment that is anticipated to expand over time through the incremental addition of third-party extensions by different software creators. One of the most challenging goals for the ecosystem coordinator and platform operator is ensuring that the introduction of new apps and developer services –or the modification of existing ones– will not create a negative impact on the platform's stability and reliability. Ecosystem coordinators see their platforms continuously growing in complexity and need specialised tools to help them control the evolution of the services they deliver, understand how changes to services can affect service consumers and ensure that services are always compliant with the variety of policies, regulations, contracts, industrial standards or technical specifications that may be applicable.
>
> Ecosystem coordinators must be able to exercise control over all critical activities taking place on the platform. Essential to achieve this is the creation of policies and policy enforcement mechanisms that facilitate governance over platform processes and resources. A platform owner's process governance policies may specify the sequence of lifecycle stages that third-party apps or developer APIs should proceed through when submitted to the platform, and the conditions for advancing from one stage to the next. Resource governance policies may specify the criteria for validating specific artefacts that are part of the submission by the third-party, such as technical specification files or service pricing information.

Viewpoint of ecosystem end-users

> On the other hand, consumers of user-facing apps and services which are delivered by the cloud application platform (i.e. end-users) find it increasingly

difficult to ensure that the services they use will satisfy all relevant requirements on a continuous basis. They too need policies and control systems to help them with governance from a service consumption viewpoint.

In this context, process governance refers to defining and enforcing policies to ensure that cloud services will be selected, tested, used and retired in a prescribed manner, with explicit conditions for transitioning from one service lifecycle phase to the next. For example, a process governance policy could state that, before an enterprise department starts using a cloud service in full scale, it must have first gone through a trial usage period, and the enterprise's Chief Information Officer must have obtained a compliance audit certificate from the provider of the service.

Conversely, resource governance refers to defining and enforcing policies to ensure that artefacts associated with cloud services being consumed conform to certain technical or business constraints. For example, a policy may state that the compliance audit certificate provided by a cloud service provider must be based on the ISAE 3402 reporting standard[24].

Viewpoint of ecosystem third-party developers

Third-party developers act as consumers of APIs provided by the platform and other developers in the ecosystem, and at the same time also act as providers of apps and services towards end-users. In that respect, their governance concerns may cover both service consumption and service delivery.

From a service consumption viewpoint, third-party developers would share similar governance requirements with end-users. A process governance policy may refer to the lifecycle followed by external services that they consume, the stages these need to proceed through and the conditions for stage transitions. A resource governance policy may place constraints on the resources associated with a service being consumed, such as specific security certificates for web APIs. From a service delivery viewpoint, third-party developers may have governance requirements similar to those of the platform owner and ecosystem coordinator.

The ecosystem coordinator is the provider of the common software platform that powers the ecosystem. In that capacity, its role may go beyond fulfilling its individual governance control objectives as outlined above, i.e. beyond only managing the quality and lifecycle of the software that enters the platform.

As we see in use cases of cloud service brokerage [10], the role of the platform owner is increasingly extending into *intermediating the governance process* so as to assist other ecosystem participants, i.e. third-party developers and end-users, with fulfilling their

[24] http://isae3402.com/

own governance requirements. To fulfil this role, platform owners are required to offer a governance support system as part of the ecosystem's common software platform, which allows different ecosystem partners to define and enforce their own governance policies [11].

2.4.3 Research on governance of software ecosystems

In relation to the main body of literature on the wider topic of software ecosystem governance, our definition of governance in cloud service ecosystems focuses on the *operationalisation of governance* rather than high level governance decision-making. It also extends beyond the single viewpoint of the ecosystem coordinator to incorporate the governance requirements of all participants in the ecosystem.

In a longitudinal survey of literature in the field of software ecosystems that includes 231 papers published between 2007 and 2016 Manikas [9] identifies three main categories under which research works can be classified: software engineering, business and management, and ecosystem relationships. Those three categories reflect the diverse focus on technology, management and social perspectives by different software ecosystems researchers, as other relevant surveys have also highlighted [65].

In the analysis by Manikas [9], the category of software ecosystems research related to software engineering revolves primarily around the theme of software architecture. The key notion here is that the architecture of a software ecosystem should support the nature of that ecosystem (i.e. be specific to its needs), support ecosystem management with business rules and restrictions, and allow the integration of diverse functionality in a reliable manner. Another theme classified under software engineering research is software quality, which relates to measuring and assuring quality of the software produced by a software ecosystem [9].

Manikas [9] identifies ecosystem governance as an emerging theme under the category of business and management research. The key notion here is that proper governance of a software ecosystem will allow ecosystem resources to be used properly, will enhance productivity and reliability and will promote ecosystem health overall.

The reason Manikas [9] classifies publications on ecosystem governance topics under business and management research rather than software engineering research, is that the vast majority of related works have so far focused on analysing or proposing models to frame governance decision-making rather than analysing or proposing models to engineer governance support systems. Focus so far has been on studying strategic governance policy-making rather than operational governance control and policy enforcement.

Serebrenik and Mens [65] carry out a meta-analysis of the research field of software ecosystems and make observations which are largely in agreement with the analysis by Manikas [9]. They present a relevant list of literature and six themes of challenges for

software ecosystems: architecture and design, governance, dynamics and evolution, data analytics, domain-specific ecosystems solutions, and ecosystems analysis.

In [66] Santos et al. undertake a review on software ecosystems as an emerging topic in the software engineering research community and outline a research agenda for the study of software ecosystems. Their research agenda comprises a total of six research themes under two different perspectives. The research theme of software ecosystem governance is classified under a management perspective, while software ecosystem quality, architecture and openness are classified under an engineering perspective.

Uludag et al. [64] develop on the ideas of Tiwana [67] and draw a distinction between governance of the software platform which represents the ecosystem's foundation and governance of actors and systems other than the platform. They refer to the first as *platform governance* and to the latter as *ecosystem governance*. The main difference between platform and ecosystem governance according to [64] is that secondary actors (i.e. ecosystem participants who can be complementors or end-users) cannot be directly controlled by the platform owner via hierarchical power or authority.

The work by Tiwana et al. in [68] and later by Tiwana in [67] has provided a thorough conceptual model of governance in software ecosystems that researchers in the field are now developing on [69],[64].

In [68] Tiwana et al. present a framework for understanding platform-based ecosystems. Their main premise is that the coevolution of the design, governance, and environmental dynamics of platform ecosystems influences how they evolve. They develop research questions to contribute towards homegrown theory about the evolutionary dynamics of software ecosystems with contributions by the disciplines of information systems, management strategy, economics, and software engineering.

In the reference book published by Tiwana in 2014 [67], titled "Platform ecosystems: aligning architecture, governance, and strategy" the overarching theme is how alignment of platform governance with its architecture shapes the evolutionary trajectory of platform ecosystems. The main premise is that architecture-governance alignment fundamentally shapes evolution, and that the two require co-designing and co-evolving through different stages of the ecosystem's lifecycle.

In Tiwana's definition, governance is about how the ecosystem coordinator influences the ecosystem, and encompasses three dimensions.

- *Decision rights:* Strategic and implementation decisions about the ecosystem's core and its complements, divided between the platform owner and third-parties.
- *Control mechanisms:* Mix of mechanisms for formal control (input, process and output control) and informal control (relational control)

- *Pricing regulation:* pricing structures, including decisions about which side gets subsidised, for how long, revenue structure.

Tiwana's view on ecosystem governance is comprehensive in that it encompasses management, technology and economics perspectives and spans across different abstraction levels from strategic governance policy making to operational governance control with policy enforcement. Another characteristic of Tiwana's view is that it is exclusively focused on the platform provider's perspective. As noted by Tiwana "*governance flows from the platform owner who governs to app developers who are governed by the platform owner*".

The vast majority of research works in the cross-disciplinary field of software ecosystem governance have so far focused primarily on the first dimension of Tiwana's model, i.e. on decision rights. The focus of this dissertation is on the second dimension: governance control mechanisms and specifically those classified under formal control.

Tiwana defines governance control mechanisms as the tools through which the platform owner ensures that the complementors' work is aligned with what is in the best interests of the platform [67]. Control mechanisms comprise the following:

- *Input control:* Also referred to as "gatekeeping". In platforms, it is common for third-party complementors who develop extensions to submit those to the platform owner for evaluation and inclusion in the platform's ecosystem and marketplace.
- *Process control:* This refers to prescribed development methods, rules and procedures that a platform owner expects third-party complementors to follow and will check for compliance.
- *Output control:* This is also referred to as "metrics-driven control" and relates to evaluating the output of third-party complementors' work. Metrics must be: (1) prespecified by the platform owner and (2) objectively measurable.

The three mechanisms above are defined as formal control. They can be complemented by an informal control mechanism that Tiwana refers to as relational control. This refers to the norms, values and culture that an ecosystem coordinator shares with complementors to influence positive behaviour and align objectives. Relational control often manifests in open-source software ecosystems and is also referred to as "clan control" [67].

As mentioned in section 2.4.2 above, in the scope of this work we consider two forms of policy-driven governance: process governance and resource governance. It is interesting to note that process control as defined by Tiwana maps naturally to our definition of process governance, whereas resource governance can be seen as a common way to operationalise input and output control as defined by Tiwana.

Tiwana notes that modularisation of the ecosystem platform architecture facilitates integration of third-party extensions with the platform only if the latter comply with the platform's interface specifications and policies. This underlines the criticality of the formal governance control mechanisms listed above to ensure compliance. However, as Tiwana notes, "testing costs are the Achilles heel of modular architectures". Automation of formal governance controls therefore becomes highly desirable.

Governance control mechanisms and specifically those classified under formal control have been well studied in other contexts and software engineering research literature but are yet to receive adequate coverage in the scope of software ecosystems research. There is however some relevant research which is worth mentioning.

Axelsson and Skoglund [70] have carried out a systematic literature mapping on quality assurance in software ecosystems. Among other topics they identify "keystone verification of extensions" which could include reliability tests carried out by the ecosystem coordinator and verification of compliance to rules and guidelines. Another topic they identify is "governance policies for verification and validation", which could include quality assurance from the viewpoint of the service consumer.

Van Angeren et al. [71] examine commercial software platform ecosystems in an inductive multiple case study to observe the entry requirements to be met by prospective app developers and the partnership and certification programs in place. Their study is among the first to empirically assess the efficacy of commercial software ecosystem governance mechanisms (input control/gatekeeping for the app store and creating partnership models).

In [43] Jansen and van Capelleveen examine methods of quality review and approval for extensions in software ecosystems. They highlight that getting third-parties to follow quality criteria and adhere to platform standards so as to provide valuable extensions is one of the greatest challenges in platform governance. They include 27 case studies and analyse them to derive three methods of governance control: review/inspection, certification, and community reviews; and eleven techniques to achieve quality goals in software ecosystems.

2.5 Governance support systems – state of the art

2.5.1 Examples of governance control mechanisms from app stores

The ecosystem coordinators who control the app stores/marketplaces of software platforms will typically enforce a submission and review/approval process for all third-party extensions. Examining the processes they follow for quality assurance and control offers useful examples of governance support systems in action – from the ecosystem coordinator's perspective.

Jansen and Bloemendal [35] provide a definition and conceptual model of app stores and survey typical features and policies observed in app stores through case studies. They define the app store as "an online curated marketplace that allows developers to sell and distribute their products to actors within one or more multi-sided software platform ecosystems". In the same paper [35] the researchers review six mobile app stores (Google Play, SlideMe, Apple Appstore, Binpress, Amazon app store for Android and Intel AppUp) to derive an overview of common features and policies. They highlight several common policies enforced by app store operators related to quality assurance and control of third-party apps: code quality review, functional quality review, interface quality review, policy compliance checking and approval before publishing. Uludag et al. [64] also compare the control measures applied by different platform providers towards secondary developers. They provide case studies in four different mobile platforms and platform ecosystems: Waze, Moovit, Apple and ITS Factory. They identify several different types of control measures applied by ecosystem coordinators including gatekeeping, regulatory checks, process control, output control and social control.

Beyond app stores for mobile platform ecosystems, making provisions for process and resource governance is equally critical for the reliability of cloud service ecosystems. This is especially pronounced for cloud application platforms allowing large volumes of third-party extensions to be deployed on a shared execution environment, physical or virtual, because of the negative cascading effects that become possible in such a setting. State-of-the-art cloud application platforms offer several examples of quality assurance and control mechanisms to govern process and artefacts.

Development and deployment of applications on Intuit Developer[25] (formerly known as Intuit Partner Platform) is described as proceeding through four phases, each of which is called "a line of development". The phases are: development, quality assurance, staging, and publishing. Similarly, on the Heroku platform[26], add-ons (i.e. third-party services) advance through the phases of development, alpha, private beta, beta, and general availability. In Force.com[27] the majority of quality checks on application artefacts are associated with a particular phase towards the end of the development and deployment process which is referred to as "security review"—though the scope of the review carried out is actually much broader than security. Progress Rollbase[28] has a similar "application approval" phase before the stage of deployment, during which all artefacts associated with an application are being reviewed against platform policies.

The policies that cloud service ecosystem coordinators specify to govern processes and artefacts are enforced through automated or semi-automated means, depending on the platform. Detailed information on how platforms like the ones mentioned above are

[25] https://ipp.intuit.com/
[26] https://www.heroku.com/
[27] https://www.salesforce.com/products/platform/products/force/
[28] https://www.progress.com/rollbase

implementing governance through policy enforcement is typically not disclosed, since they are commercial offerings. In general, some platforms provide online tools that span the entire process of creating and deploying new services and applications, while others provide offline tools for development and testing (e.g. as popular IDE extensions) and employ certain online tools only for submitting services and applications to the platform and initiating their deployment. These tools will often support automated artefact validations (e.g. XML schema validations) as means of artefact-level policy enforcement, and less often, may also support some form of transition eligibility checks before applications can be promoted from one phase to the next.

Given the relative immaturity of the domain, standardised specifications and software solutions for governance support of cloud application platforms have not yet emerged. Therefore, many of the tools that each platform employs for implementing process and resource governance are expected to be custom, purpose-built one-off solutions. This concerns both externally-facing tools, i.e. tools for third-party developers, and internally-facing tools, i.e. tools used by the platform providers' own administration and quality assurance staff.

Nevertheless, for many of the tasks associated with process and resource governance in the context of cloud application platforms, useful lessons may be learned from the related field of service-oriented architecture (SOA) governance [72],[73], from which mature solutions may also be transferred. The two have many things in common. Fundamentally, both are employed to deal with the complexity involved in managing loosely coupled, independently developed, and dynamically aggregated units of software. As stated in the OASIS Reference Architecture Foundation for SOA [74], "owning a SOA-based system involves being able to manage an evolving system'" This outlook is consistent with the earlier discussed view of cloud application platforms as environments of ever-increasing size and complexity.

2.5.2 Best practices from SOA governance

Not too many years after service-oriented architecture (SOA) was introduced as a term in the late 1990s, adopters started realising that without appropriate governance over the various phases and activities associated with the lifecycle of distributed services, a SOA-based IT infrastructure can quickly dissolve into an unmanageable environment [60],[75]. Since then, governance has become broadly recognised as a precondition for the success and long-term sustainability of service-oriented architectures, and as a major challenge, from both a decision-making and operationalisation perspective [59].

As mentioned earlier in this chapter, the challenge of governance from the viewpoint of operationalisation lies in providing effective and efficient support for the daily activities of stakeholders in a SOA-based computing environment, such that governance imperatives, expressed as policies, can be enforced in a transparent and

preferably automated way throughout the lifecycle of all services. A best practice that has been established over the years is to address this challenge with the help of governance support systems that integrate registry and repository functions [76].

In an analogy with the way yellow pages are used, a registry system allows providers and consumers of software units (such as web services in the case of SOA, or cloud apps and developer APIs in the case of cloud application platforms) to maintain a catalogue of the services available. In the case of SOA, the entity that maintains this type of registry is typically the owner of the SOA-based computing infrastructure, with the collaboration of internal and external developers of service-based systems and applications that utilise it. Correspondingly, the interested parties in cloud application platforms are first and foremost the platform provider, but also third-party developers who build extensions to the platform in the form of new developer services and cloud apps. Every new software component on the cloud application platform is registered and is given a description that other parties can use as reference. The description consists of metadata providing information about the component, definitions of the component's relationships to other components, and associations to any relevant artefacts.

Repository systems are complementary to registries, as they offer the means for storing and managing the actual artefacts that may be associated with a registered software component. Those artefacts may be specific to a single registered component or associated to more than one. Notably, a single change in an artefact may cause significant changes to the state of other dependent artefacts, or to the states of associated services and applications. Artefacts within a repository should therefore be managed and monitored in a way that allows tracking changes, detecting dependencies, and analysing the impact that a change can have in order to take appropriate measures. In general, storing all artefacts that a software component comprises in a central location enables a systematic approach to access control, versioning, dependency tracking, change management, and policy enforcement.

The purpose of a combined registry and repository system within a service-based infrastructure is to provide an authoritative system of record. Governance support systems couple this with a set of functions supporting governance of different types of entities and artefacts through the definition and enforcement of policies.

Registry and repository systems are typically found at the core of every commercial governance technology platform. Gartner Research has recently surveyed the market for application services governance solutions in a special technology analysis report [77]. The report reviews the capabilities of commercial governance support systems by IBM[29], SAP[30], Mashery[31], Apigee[32], Axway[33] and others, some of which follow a

[29] https://apim.ibmcloud.com/
[30] http://scn.sap.com/community/api-management
[31] https://www.mashery.com/
[32] https://apigee.com/about/cp/api-governance

commercial open-source model such as Mulesoft[34] and WSO2[35]. Gartner highlights that "*the growth of the API economy, the pressing need for application rationalisation and the disruptive needs of digital business applications will continue to change the market for application services governance technology*".

2.5.3 Definition and enforcement of governance policies

Governance support systems provide users with some way of checking whether the governed resources, as described by the data held in the registry and repository system, conform to relevant policies. Each solution achieves this through a different approach to policy definition and enforcement.

In the scope of the research carried out within project CAST [22], the author of this dissertation analysed and compared the commercial open-source governance support systems by Mulesoft and WSO2 – the two open-source solutions which are also highlighted in Gartner's industry survey [77]. The purpose of the analysis was to understand how the two systems allow governance policies to be defined and enforced [78]. The results from the analysis of the approaches adopted by the two solutions are discussed in detail in chapter 4. In the interest of introducing the state of practice regarding policy-driven governance support systems some key aspects are briefly discussed here.

The analysis revealed some characteristics which represent limitations that are especially pronounced in a cloud service ecosystem context. These limitations fundamentally stem from how policies are defined and enforced, and can be summarised in the following.

- *Lack of separation between definition and enforcement of policy*: Policy definition and policy checking are entangled within the same software unit. Policy authors write custom code that interfaces with the governance support system through an API, and checks if some data of interest conforms to certain constraints. Those constraints are not set out as explicit self-contained policies, but defined implicitly as part of the same code that checks for data conformance. Except for the case where such constraints are defined in an explicit way (e.g. in a separate XML schema document) there is no differentiation between *what* a policy is about, and *how* data can be checked for conformance to that policy.
- *Lack of abstraction in policy representation*: Because of the above, policy logic is represented at the same level of abstraction as the implementation of the governance support system. The rules or constraints that a policy comprises are encoded in an imperative style, as part of the same low-level logic that queries

[33] https://www.axway.com
[34] https://www.mulesoft.com/resources/esb/application-service-governance
[35] http://wso2.com/products/governance-registry/

databases and parses files to check instance data for violations. The encoding of a policy is therefore disconnected from the high-level domain concepts that one would use to communicate its purpose and the policy author's intent.

- *Lack of formal representation of policy rules and relationships*. The relationships among policies, as well as between policies and their subjects (i.e. the logical entities in the governance domain) are not captured explicitly. Tracing the association of an operational-level policy to other policies at the same level or a higher (strategic) level is not possible. The same holds for tracing the relationships between a particular platform resource and all policies directly or indirectly related to it. Last but not least, the absence of any formal encoding of policies makes it difficult to analyse them, to reason how policies may affect other policies and to perform automated verification and validation. Typically, the only machine-readable representation of a policy is the code that enforces it.

As we will discuss later in chapter 3, the above characteristics have negative implications with respect to policy maintainability, comprehensibility, verifiability, traceability, interoperability, and with respect to the overall agility of a governance support system.

Overall, there is lack of support for decentralised/distributed cooperation. State-of-the-art governance support systems have been created with centralised governance in mind. Their design is driven by the notion that there is some entity which needs to make sure certain policies are enforced (for its own benefit, not as part of providing a service to some other entity in an ecosystem). To achieve this, the central entity (typically, an enterprise) will both create the policies and enforce them. To create and enforce those policies this entity obtains or creates descriptions of the governed resources and stores them in a central location (registry and repository system) which is owned, hosted and operated by the same entity.

This approach is well suited to serve the individual platform governance needs of an ecosystem coordinator, but cannot simultaneously accommodate the governance needs of other ecosystem participants (i.e. third-party developers and end-users) as discussed in section 2.4.2. In an intermediated ecosystem governance process, there could be several different entities with the need to have their policies enforced. And these policies may not be defined by the same partner that enforces them, such as the ecosystem coordinator, but possibly from other ecosystem partners. The same holds with the governed resources and their descriptions which are subject to governance – they may be provided by an ecosystem partner who is different to the one evaluating the policies. These different roles and concerns of stakeholders in the governance process are discussed in the chapter that follows.

2.6 Summary

In this chapter we introduced the background to the problem domain addressed by this dissertation. We first introduced the paradigm of cloud computing and provided definitions for the most commonly used cloud computing service models: Software as a Service (SaaS), Infrastructure as a Service (IaaS) and Platform as a Service (PaaS). We next focused on the model of PaaS and discussed cloud application platforms in terms of the benefits they offer to application developers. In broad terms, cloud application development platforms which are delivered as PaaS offerings could be classified as domain-agnostic or domain-specific platforms and they may follow quite different architectures to allow extensions by third-parties.

In the section that followed we focused on the co-development possibilities and ecosystem creation potential presented by cloud application development platforms. We introduced software product co-development as a growing trend and the different basic or advanced co-development models that may manifest depending on the architecture of a platform.

Software platforms which facilitate co-development relationships between different partners foster the creation of environments best characterised as software ecosystems. We introduced the foundations of software ecosystems and adopted the definition by Jansen et al. [5] who define software ecosystem as "*a set of businesses functioning as a unit and interacting with a shared market for software and services, together with the relationships among them. These relationships are frequently underpinned by a common technological platform or market and operate through the exchange of information, resources and artefacts*." We also discussed the primary roles that emerge in a software ecosystem setting.

Analysing the definition of software ecosystems by Jansen et al. [5] we can recognise the same characteristics in environments where the "common technological platform" is a cloud application platform. That leads us to introducing a special class of software ecosystems we refer to as cloud service ecosystems. To illustrate this notion we summarised the key characteristics of four commercial cloud service ecosystems: Microsoft Azure, Heroku, Force.com and SAP YaaS.

In the following section we introduced the challenge of governance. We provided definitions for what is a rather broad and misused term and explained the positioning of our research relative to governance as a theme in the literature. We then narrowed focus on cloud service ecosystem governance and discussed what it can mean to different types of stakeholders. We provided our own working definition of governance in cloud service ecosystems, which refers to the process and the supporting systems for defining and enforcing policies to control the creation, provision and consumption of cloud services by different ecosystem partners.

We then provided an overview of the related body of literature on governance of software ecosystems so as to position our work relative to other software ecosystems research. We expanded on Tiwana's definition of software ecosystem governance which encompasses the dimensions of decision rights, control mechanisms and pricing regulation. We highlighted that the vast majority of research works in the field of software ecosystem governance have so far focused on the first dimension, i.e. on decision rights, while the focus of this dissertation is on the second dimension: governance control mechanisms and specifically those classified under formal control. In relation to the main body of literature on the wider topic of software ecosystem governance, our definition of governance in cloud service ecosystems focuses on the *operationalisation of governance* rather than high level governance decision-making. It also extends beyond the single viewpoint of the ecosystem coordinator to incorporate the governance requirements of all the participants in the ecosystem.

To provide context, we presented examples of governance control mechanisms from app stores of cloud service ecosystems. We then moved to discussing the state of practice for creating and operating governance support systems. We introduced best practices from SOA governance and discussed some of their characteristics relative to how policies are defined and enforced, so as to show that these represent important limitations in the context of governance for cloud service ecosystems.

The key takeaways from this chapter can be summarised as follows:

1. Increasingly, cloud application platforms follow an open architecture which allows third-parties to enrich a platform's capabilities with their own add-ons. This collaboration model leads to a form of software product co-development. Platforms that facilitate co-development relationships between partners foster the creation of environments best characterised as software ecosystems. Cloud service ecosystems can be seen as a special class of software ecosystems.
2. The value of an ecosystem increases exponentially with more users and more complementary services. But complexity is a threat to system reliability. Ecosystem partners need to be able to exercise control over developments in the ecosystem that may affect them, such as the introduction of a new service, a change to the characteristics of an existing service, or a change to how services are assembled. This is a challenge of governance.
3. In all its different expressions, governance is fundamentally a cross-disciplinary subject that can be viewed from both an organisational and technological perspective. The majority of researchers on the subject focus on the first viewpoint, i.e. on how governance decisions can be made in an organisation. In this work we focus on the latter viewpoint, placing emphasis on the policy-driven control mechanisms that are necessary to operationalise those governance decisions. Our focus is on how to create software systems that support policy-driven governance.

4. We define governance in cloud service ecosystems as the process and the supporting systems for defining and enforcing policies to control the creation, provision and consumption of cloud services by different ecosystem partners. In relation to the main body of literature on the wider topic of software ecosystem governance, our definition extends beyond the single viewpoint of the ecosystem coordinator to incorporate the governance requirements of all participants in the ecosystem.
5. Examining state-of-the-art governance technology platforms reveals a gap between the type of requirements these platforms were originally designed to meet and the type of needs emerging to support governance in this new context of software ecosystems. The most prominent shortcoming is lack of support for networked collaboration. Contemporary governance support systems have been created with centralised governance in mind. Their design approach is well suited to serve the individual platform governance needs of an ecosystem coordinator, but cannot simultaneously accommodate the governance needs of other ecosystem participants, i.e. third-party developers and end-users.

Chapter 3

Governance in cloud service ecosystems: key requirements

3 Governance in cloud service ecosystems: key requirements

3.1 Introduction

As discussed earlier in this dissertation, cloud service co-development ecosystems are complex environments composed of many participants, each of which may have their own view of the ecosystem, their own objectives from participating, and their own concerns. This encompasses all aspects of their function in an ecosystem, including their participation in governance processes.

From the viewpoint of the platform owner and ecosystem coordinator, governance is a process ensuring that introducing new software to the platform will not create negative ripple effects. From the viewpoint of ecosystem participants, i.e. third-party developers and end-users, governance is a process ensuring that the services they consume or deliver will continuously operate as expected and will satisfy requirements on an ongoing basis.

This chapter introduces the idea that software systems which are aimed at supporting the governance needs of ecosystems need to achieve an appropriate separation of concerns based on the different types of roles engaged in governance processes.

To understand what this entails in terms of the concrete qualities that a governance support system should exhibit, one first needs to understand the different types of concerns associated with governance process stakeholders. The following sections in this chapter aim to assist the reader in this direction.

The motivation for the analysis presented in this chapter, as well as the source of the insights that underpin the idea of separating concerns when designing governance support systems, surfaced during the author's work in research projects CAST[36] and Broker@Cloud[37]. Both of these projects included an objective to create software systems supporting governance in cloud service ecosystems. My work in these projects offered the opportunity to gather and analyse a variety of governance process scenarios. Analysis showed that any process of governance in the context of a software ecosystem is inherently a collaborative (and typically inter-organisational) process. Analysis also led to observing a recurring pattern in governance processes – a small set of distinctive roles and typical interactions was always present in the process. Looking closer at the needs fulfilled by these roles resulted in conceptualising a core set of concerns associated with each type of governance process role.

[36] http://seerc.org/projects/cast/
[37] http://www.broker-cloud.eu/

To illustrate the complexity and diversity of governance processes and stakeholder roles in full variance we have chosen to synthesize example scenarios that combine multiple different elements from governance processes found in actual operational environments. Section 3.2 of this chapter presents five such exemplifying scenarios of governance in a cloud service co-development ecosystem inspired by real-world use cases, including use cases derived from industrial partners in research projects CAST and Broker@Cloud.

The examples demonstrate governance processes ranging from relatively simple to fairly complex. We use these example stories to introduce the discussion that takes place in the next section of the chapter (Section 3.3) on key roles of stakeholders in a governance process, their views of the process, their associated concerns and individual objectives. This discussion leads to highlighting some fundamental design requirements for governance support systems, at the end of this chapter (Section 3.4).

3.2 Examples of governance in a cloud service ecosystem

This section presents five example scenarios that serve to illustrate potential forms of policy-based governance in the wider context of a cloud service ecosystem. Each example demonstrates a different potential scenario of policy-based governance in action. Each story brings different aspects of governance to the foreground, making implicit links between the roles and processes in the ecosystem.

3.2.1 Scenario 1: Quality review in a private PaaS environment

NineLives is a multinational medical insurance company with business operations across several European countries. To reduce the costs associated with maintaining disparate legacy information systems for its operations in different locations, the company has decided to rebuild its entire business process support infrastructure around a single, common software platform. The platform software was licensed from CloudDev, a company that provides software and services for the development and hosting of on-demand business applications. CloudDev offers subscription-based access to its platform as a public cloud service, but instead of that option, NineLives reached an agreement with CloudDev allowing it to run the platform software of CloudDev on the privately owned data centre of NineLives.

The legacy systems deployed at different divisions of NineLives are gradually being replaced by cloud applications which are deployed and hosted on the common platform environment. These cloud applications are developed by different teams. Some applications are developed by internal staff at various IT departments of NineLives in different branches, while other applications are developed by external partners (outsourced).

To ensure that all these different applications can be smoothly integrated into a common operational environment, the team that manages the private PaaS environment at NineLives has established a number of policies that application developers are required to follow. These policies affect many different aspects of an application, such as how to use external and internal platform APIs from within an application, how to structure an application's deployment descriptor, how to provide administrative and technical contact details for the team supporting an application, how to use logos and other visual design elements, etc.

Once the development and testing of an application is complete, developers submit the application to the NineLives platform management team for quality review. Quality review personnel undertakes to make sure that every application conforms to platform policies before it is allowed to become operational. When policy violations are identified these are reported and the application is not allowed to proceed to deployment. To check applications against company policies the quality review team is using a combination of manual and automated policy evaluation methods. The process is as follows. First, the quality review staff determine which policies are applicable to the application at hand (different policies apply to different types of software applications). Second, they collect all the relevant data from the different artefacts that come with the application. Third, the collected data is evaluated against the conditions set out in the relevant policies. At the end, the quality review staff completes a quality review report that summarises the results, and whether or not policy violations have been detected.

The scenario is illustrated in Figure 3 below.

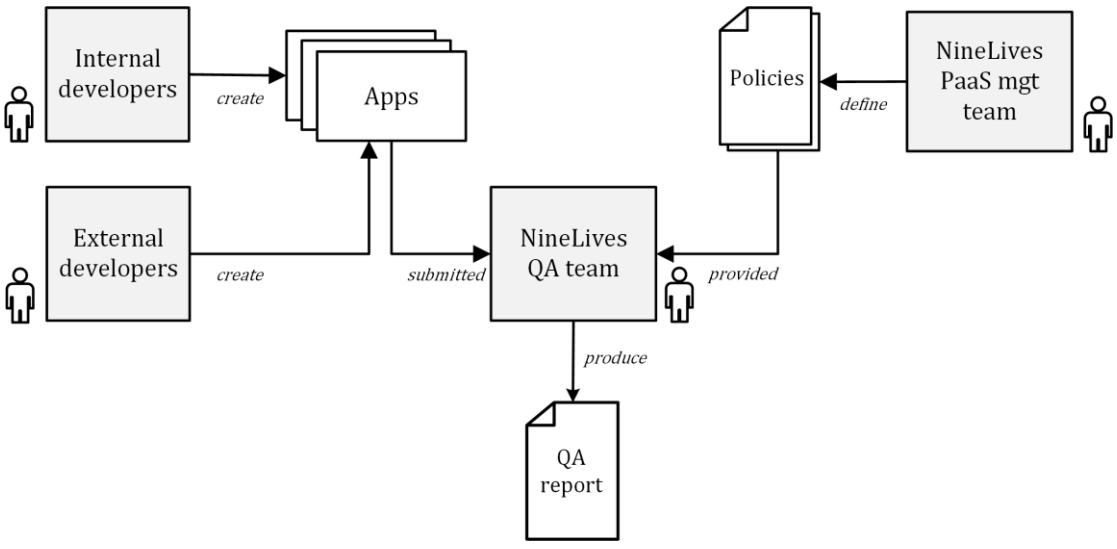

Figure 3. Scenario of quality review in a private PaaS environment

3.2.2 Scenario 2: Regulatory compliance audits of cloud applications

NineMed is a partly-owned subsidiary of NineLives, incorporated in a different European country. As a publicly-traded company with activity in the highly regulated domain of medical insurance, NineMed is required to comply with a wide array of regulations governing its operations. These regulations are often imposed by different external authorities such as the national government and the European Union. In addition, as a subsidiary of NineLives, NineMed is required to comply with a number of "internal" policies set by the parent company.

NineMed needs to be fully aware of all the different regulations it is required to observe, and bears the responsibility to comply in a proactive manner. The company is subject to external and internal audits to ensure that this is the case. Ensuring compliance is the responsibility of the company's chief compliance officer (CCO) who is in charge of the company's compliance management team. The compliance management team is tasked with monitoring the regulations published by different authorities, the policies issued by the parent company and the daily business practises of NineMed, codifying all this policy information in an internal knowledge base and reporting compliance issues to upper management.

To perform its function, the compliance management team at NineMed first needs to identify which business processes and company information systems are affected by any particular regulation or policy under consideration. Then, it needs to obtain details on how the affected business processes are carried out within the company, and how the associated information systems operate. This information needs to be retrieved from several different resources within the company, such as process guidelines, staff manuals and software documentation. Next, the compliance management team needs to evaluate the collected information against the observed policies. In case of non-compliance the chief compliance officer is required to report this and the company is expected to take immediate rectifying action.

In advance of the company's migration from its old systems to new on-demand applications developed against the CloudDev platform, the compliance management team is carrying out compliance audits on the new applications as these are turned in by its IT staff. The goal of the audits is to ensure that the new applications comply with all relevant external regulations and internal policies. Sometimes the audits reveal problems associated with the fact that rules are issued by different authorities. It often turns out that some of NineMed's local policies are in conflict with global policies originating from NineLives - the parent company, or that policies imposed by the parent company are not in line with local legislation.

The scenario is illustrated in Figure 4 below.

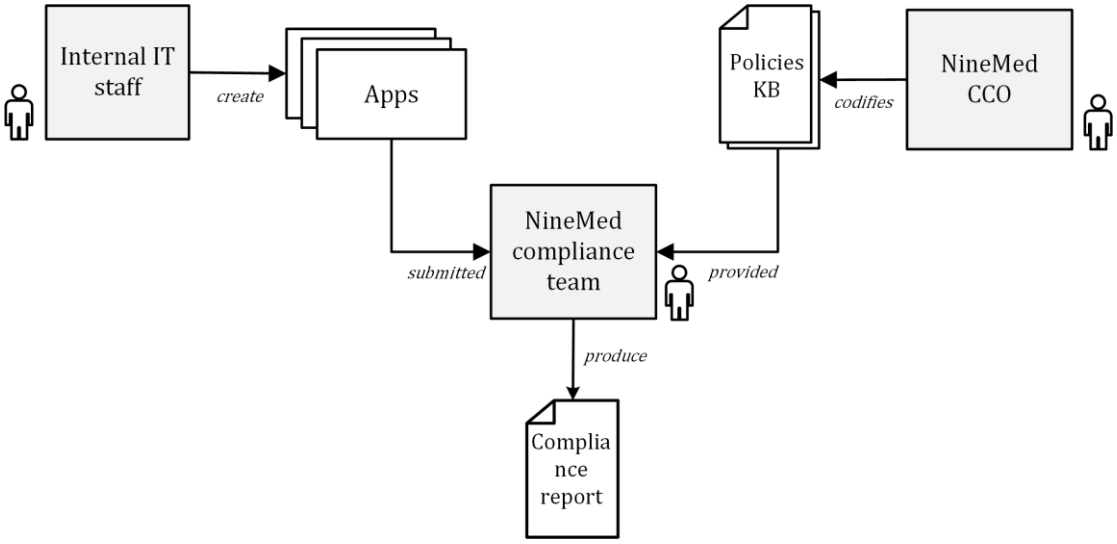

Figure 4. Scenario of regulatory compliance audits of cloud applications

3.2.3 Scenario 3: Lifecycle management and quality control in a cloud service ecosystem

In addition to licensing its cloud platform for private deployment to large customers such as NineLives, CloudDev also offers access to a hosted instance of its platform which is being managed by CloudDev itself. The company markets this service as an application Platform as a Service (aPaaS) offering. Through this offering, the company has managed to become the facilitator of an ecosystem of cloud services by third-party independent software vendors (ISVs) who develop, host and market their applications in collaboration with CloudDev. One of the things that attracted those ISVs to the ecosystem was the built-in capabilities of the CloudDev platform for a range of generic business application functions, such as document management, content authoring and collaboration. Making use of these capabilities allows ISVs to develop small-scale solutions for small and medium sized businesses with minimal effort and resources. The second important factor attracting ISVs to CloudDev is that the company offers a free marketing and distribution channel through the CloudDev application store. Revenues from monthly subscriptions to ISVs' applications are shared between the ISVs and CloudDev.

Apptitude is a small ISV that specialises in SaaS applications for the Human Resource Management industry. The company is part of CloudDev's ecosystem, but also develops and markets its applications through a number of similar (and mutually competing) cloud application ecosystems that are oriented towards business solutions for small and medium sized businesses. Most of Apptitude's applications are multi-homed[38], meaning that the company has built several variations of them to allow deployment on different cloud platforms, and distribution through several different app stores and marketplaces.

[38] For a discussion on multi-homing in software ecosystems the interested reader is referred to [79]

When Apptitude creates a new application for the CloudDev platform it needs to proceed through the steps of a formal lifecycle management process imposed by CloudDev on all ISVs. The process has been put in place to prevent the adverse effects of introducing a problematic application into the ecosystem, and to ensure that the environment remains healthy and competitive. During the application development phases, unit and integration testing take place in an isolated development environment – a development sandbox. Once this stage is completed ISVs can choose to launch a private beta testing programme with a limited number of invited users. This is carried out in an isolated trial environment – a beta sandbox. When a new application is finally ready for release Apptitude submits the final version of its codebase and the application description to CloudDev. The artefacts that comprise the application need to observe a number of policies set out by CloudDev. These concern both technical aspects such as restrictions on application coding standards or how applications use platform resources, as well as business aspects such as restrictions on an application's pricing model.

Before the application codebase is deployed onto the CloudDev production environment and the application description is added to the CloudDev application store, a quality review step takes place. The CloudDev quality assurance staff examines the code and metadata submitted by Apptitude and employs a combination of manual and automated methods to ensure that all relevant policies are observed. In case of policy violations these are reported and the release is blocked. Alternatively, the application is allowed into the main production environment and into the application store. Consumers can thereafter select the application and subscribe to use it. When Apptitude wishes to retire an application there is another set of conditions to be checked. CloudDev's main objective here is to ensure that decommissioning an application does not have any adverse effect on consumers who are still using it and on other applications that are interfacing with the application to be retired.

The scenario is illustrated in Figure 5 below.

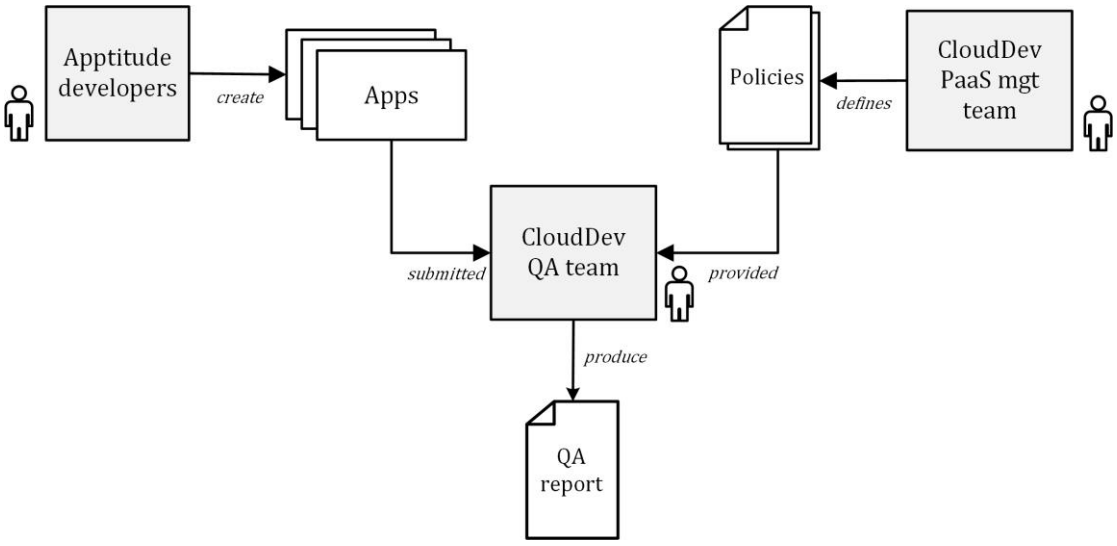

Figure 5. Scenario of lifecycle management and quality control in a cloud service ecosystem

3.2.4 Scenario 4: External auditing of cloud service providers

Better, Saffe & Sawrie is an auditors' firm specialising in Governance, Risk and Compliance (GRC) services for the hi-tech industry. Among other things, the company offers external auditing services to providers of cloud services, helping them to demonstrate compliance to standards such as ISO 27001 or SSAE 16. The audits carried out by the firm may involve infrastructure (IaaS), platform (PaaS), or application (SaaS) services.

Compliance to standards is evaluated through review of objective evidence, which is provided by the customer in accordance to the auditor's specifications. This involves a wide range of data about the customer's information systems, business processes and operations. Once collected, the data is evaluated by the auditor to validate that the customer's IT environment has all the appropriate mechanisms in place as foreseen by the customer's own policies or by third-party standards and to verify that these mechanisms operate as intended.

It is common for cloud service providers to subject their software and hardware infrastructure to periodic external audits to offer better assurances to customers about the safety of their data, their privacy, and the overall reliability of their services. Being able to offer such assurances and proof of compliance to major industry standards has become a necessity in the cloud computing market.

Compliance certification of cloud service providers is especially sought after by consumers that are publicly-traded companies. Law holds officers of publicly-traded companies responsible for the quality of their company's financial statements. The quality of those statements is affected not only by a company's internal controls but also by the internal controls of third-parties who provide operation-critical services to a company. For this reason, a publicly-traded company that uses cloud services from a third-party will have to either carry out an audit of internal controls at the cloud service provider —which is something that the latter would most likely refuse— or to ask the provider to present a compliance audit certificate produced by an independent agency.

One of the cloud service providers that receives periodic external audit services by Better, Saffe & Sawrie is CloudDev. The auditing process proceeds as follows. Firstly, CloudDev informs the auditing agency about its compliance objectives, meaning the internal policies it wishes to observe, and how it has chosen to implement public cloud security standards on its infrastructure. Subsequently, the auditor examines the compliance requirements, and issues a checklist of sample artefacts to be collected as well as reports to be filled-in by key personnel at CloudDev. CloudDev collects the data as per the auditors' specifications. This may include information about the security controls of the aPaaS platform, its mechanisms for data storage, for system activity logging and monitoring, for redundancy and availability, and much more. The auditors examine the submitted data and may also carry out on-site inspection visits. The audit

results are summarised in a service auditor's report. If the audit is successful, Better, Saffe & Sawrie issues an audit certificate.

The scenario is illustrated in Figure 6 below.

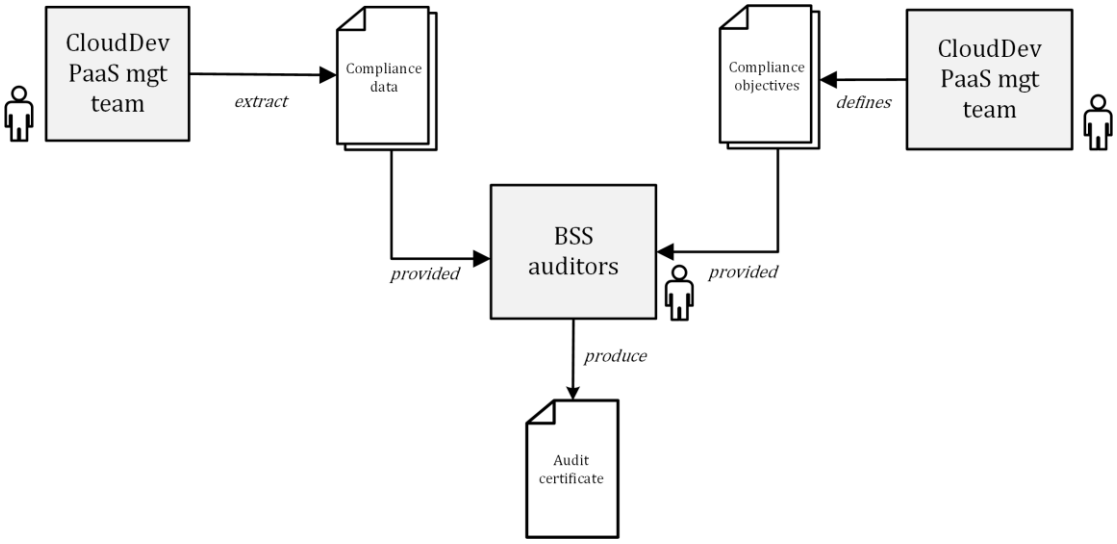

Figure 6. Scenario of external auditing of cloud service providers

3.2.5 Scenario 5: Policy-based governance by a cloud service broker

Appregator is a cloud service broker that helps consumers engage with the right cloud services provider for their needs. It maintains an extensive directory of cloud services across a number of areas and allows side by side comparison of offerings. In addition to helping with service selection Appregator helps consumers to manage their service contracts on a continuous basis. This is achieved by three means. Firstly, by taking over the task of monitoring the performance of providers against the Service Level Agreements (SLAs) agreed with the consumer. Secondly, by monitoring the conformance of providers to custom policies specified by the consumer. Thirdly, by monitoring the cloud computing market and offering recommendations to the consumer about alternative services based on customer-specified optimisation criteria.

To be able to offer an accurate picture of the cloud services available in the market and how these services compare, Appregator collects data from several cloud service marketplaces on a continuous basis. It retrieves the descriptions of the cloud services from distribution channels, reconciles the information to an internal information model that facilitates homogenisation and comparison, and adds the resulting descriptions to its service directory. Service descriptions cover a wide range of aspects, such as key features, technical characteristics, subscription costs, terms of service and SLAs. Based on this information consumers can narrow down search and comparison to specific attributes of a service, they can specify custom policies to be monitored, and can fine-tune preferences for optimisation.

One of Appregator's customers that have chosen to outsource the management of their contracts to the cloud service broker is Cerebrate; a medium-sized Human Resources Management (HRM) agency helping companies in the financial services sector with sourcing, screening, evaluating and recruiting employees. Cerebrate has been trying out different SaaS applications to help the agency's staff with identifying candidates on social media, matching them to job positions, planning interviews and evaluation tests, and managing the entire recruitment process. Appregator has helped the agency to save considerable time and effort in identifying services suitable to its needs, and Cerebrate has decided to also let Appregator manage its ongoing subscription contracts for some of those services.

The management of Cerebrate's cloud service subscriptions encompasses several aspects. As part of its work Cerebrate is dealing with personal data of candidates as well as sensitive business information disclosed to the agency by the companies it recruits for. Confidentiality, privacy and safety of data are therefore important parameters in engaging with a cloud services provider. Cerebrate has established a set of policies about how its services providers should address these issues (for instance, strong encryption measures) and has outsourced the monitoring of cloud service conformance to Appregator. Another important parameter is business continuity. Having outsourced many of its IT functions to external service providers, the agency's daily operations are entirely dependent on the availability and performance of those services. To ensure appropriate levels of operational capacity Cerebrate has established a number of related policies (e.g. an upper limit on the time it should take for a service to recover from a failure) which are enforced by Appregator. Appregator monitors changes to the terms of service in the cloud services contracted by Cerebrate, and ensures that these continue to conform to the agency's policies at all times. Yet another important parameter is cost. To keep operational costs under control Cerebrate has fixed upper limits on the usage of cloud services by its staff. Appregator monitors billing information in real-time and issues alerts when service usage exceeds the monthly limits in the agency's policy.

One of the cloud services that Cerebrate has been using is TalentForge by Apptitude. TalentForge is a SaaS web application helping HR professionals with candidate evaluation and employee performance assessment. Apptitude has created multiple versions of the TalentForge application to allow distribution through multiple channels (multi-homing). This allows Apptitude to be part of several cloud service ecosystems at and to leverage as many synergies with other cloud service providers as possible.

The version of TalentForge that Cerebrate has been using is the one developed on top of CloudDev's aPaaS platform. CloudDev is one of the many distribution and delivery channels that Appregator is monitoring. Likewise, Appregator is one of several cloud service brokers that CloudDev collaborates with to promote its ecosystem. CloudDev encourages intermediaries (brokers) to retrieve and republish the service-related information available on the CloudDev app store, as this increases visibility for its

services and drives web traffic and subscribers to its marketplace. To make it easier to retrieve information about the SaaS services developed by the ISVs in its ecosystem, CloudDev provides special APIs for intermediaries to use. Moreover, CloudDev provides real-time data feeds with detailed information on consumer-specific service usage, such as billing events, which intermediaries such as Appregator can monitor on behalf of the end service consumers such as Cerebrate.

To manage the TalentForge subscription contract that Cerebrate has in place with Apptitude, Appregator relies on the data exchange mechanisms that CloudDev has made available. The service contract management process comprises the following activities. As a first step, the cloud service consumer —Cerebrate, in this case— needs to let Appregator know which service contract needs to be managed. Appregator will allow Cerebrate to specify the aspects of the agreed service contract that should be monitored (e.g. service performance attributes), to define custom policies on service properties of importance (e.g. security and reliability attributes), and to set optimisation criteria as triggers for generating recommendations. Next, Appregator will undertake to monitor changes in CloudDev's app store that concern TalentForge. Every time something changes Appregator will evaluate whether this is relevant to the requirements of Cerebrate, and whether or not it raises a conformance issue. In parallel, Appregator will undertake to evaluate developments in its entire directory of services and whether or not these represent optimisation opportunities for Cerebrate, such as replacing TalentForge with a competitive offering.

The scenario is illustrated in Figure 7 below.

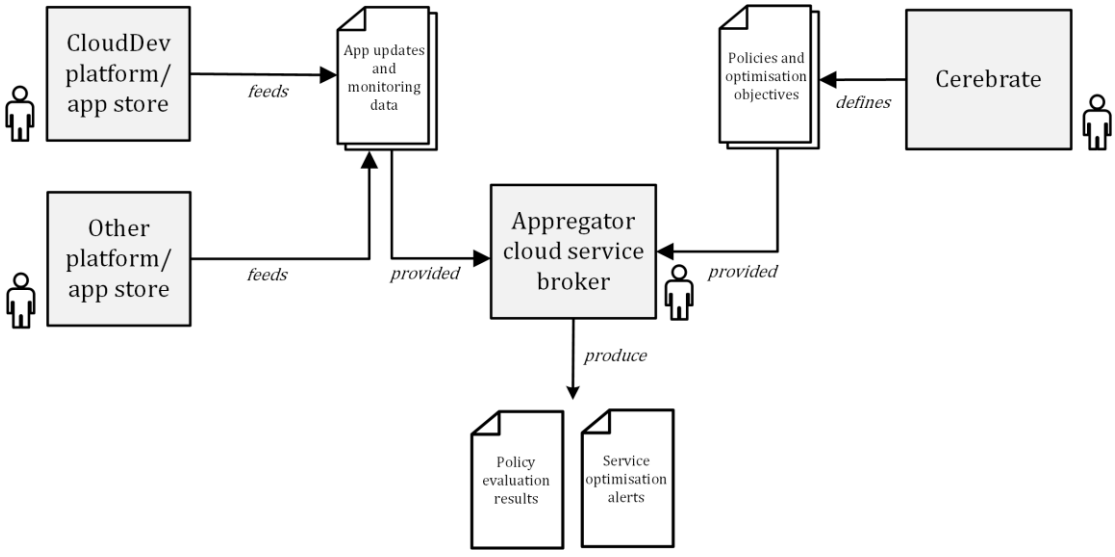

Figure 7. Scenario of policy-based governance by a cloud service broker

3.3 Roles and concerns of stakeholders in the governance process

Implicit in the scenario examples presented in the previous section are a number of roles and functions which are common in cloud service ecosystem governance scenarios. This section makes these features explicit. The goal is to highlight some fundamental concepts relating to governance in cloud service ecosystems, and to lay the groundwork for requirements thinking on governance support systems.

3.3.1 Roles in the governance process

Examining the scenario examples reveals three recurring roles which interact in the governance process. These are:

- ***Policy provider role:*** Responsible for creating, maintaining, and providing governance policies that need to be enforced. A policy may affect more than one governed resources in a software ecosystem. Different policies by the same provider may reflect obligations at different levels of abstraction (e.g. strategic-level vs. operational-level policies) and may have different origin (e.g. internally defined corporate policy vs. externally imposed regulatory framework). An example of this role from scenario 1 is NineLives, the life insurance enterprise.

- ***Data provider role:*** Responsible for creating, maintaining, and providing data describing the software resources in the ecosystem which are subject to governance. This data is necessary in order for the different policies affecting those resources to be evaluated. Governed resource descriptions could be derived from primary data residing in files or databases, or data derived from primary sources specifically for the purpose of policy evaluation. An example of this role from scenario 3 is Apptitude, the HR solution developer.

- ***Policy evaluator role:*** Responsible for creating, maintaining, and providing a software system or software-based service that facilitates governance by carrying out policy evaluation. That is, a system or service that checks whether a certain governed resource description conforms to a specific policy or not. The policy of interest and the information about the governed resource of interest are provided as inputs to the evaluation process. An example of this role from scenario 5 is Appregator, the cloud service broker.

3.3.2 Distribution of governance roles

Governance in a cloud service ecosystem can be a distributed process. Actors in the ecosystem may simultaneously assume more than one of the governance roles

mentioned above, while at the same time, there can be more than one actor acting in the capacity of a certain governance role.

For example, in scenario 1 there are two stakeholders who act as Data Providers - the internal and external developer teams (see Figure 3), while in Scenario 2 where there is one stakeholder who directly provides the policies in codified form (the CCO) we can understand there are more stakeholders inside NineMed and its parent company who indirectly also act as Policy Providers (see Figure 4).

At the same time in Scenario 3 an entity – CloudDev – acts as both Policy Provider and Policy Evaluator (Figure 5), while in Scenario 4 it is the roles of Policy Provider and Data Provider that coincide on CloudDev as the actor (see Figure 6).

3.3.3 Types of concerns

Associated with each role in the governance process is a distinct set of concerns and goals, which the respective stakeholder(s) would like to see satisfied.

Each entity that engages in the governance process of a cloud service ecosystem has a business goal to minimise cost and maximise benefit. This results in governance roles adopting concerns at two different levels of abstraction.

- **Role-level concerns** which are local in scope, limited to the specific governance role, and involve internal management of the governance process from the perspective of that role.
- **Ecosystem-level concerns** which are global in scope, seen from a wider perspective, and involve external collaboration with other roles in the governance process.

3.3.4 Policy provider concerns

The primary concerns of any actor assuming a policy provider role are manageability and evolvability of their internal function/process.

The first means to be able to effectively and efficiently manage policies, which includes not only creating, testing and maintaining the policies, but also managing the knowledge regarding how different policies interact, or relate to each other.

The latter means to be able to change and evolve its internal process for policy provision, and effect any desired changes without disrupting the operation of other ecosystem actors who need to comply with or enforce policies. And conversely, to avoid unnecessary disruption from changes effected by other ecosystem actors.

Manageability of the policy provision function:

- *Making it easy to create, test and maintain policies*: For instance, can the policy encoding language be easily understood by the domain experts who author the policies? Does it allow defining policies on the basis of their domain vocabulary? Is it possible to construct policies by modular composition, by reusing existing policies? Are the policies encoded in a policy language which is amenable to automatic analysis? Is it possible to automatically check for policy self-coherence (internal policy conflicts) and "debug" policies?
- *Making it easy to manage knowledge about policies*: Being a policy author creates a knowledge management requirement, as there may be important information regarding the policies which is not captured in the policy encoding itself. This implicit knowledge needs to be made explicit. For instance, do the policy language and the tools used for policy encoding support capturing policy hierarchies? Do they allow capturing relationships between policies at various levels of abstraction so as to support policy dependency analysis and inter-policy conflict detection (conflicts between policies by the same provider or by different providers)? Do they allow answering questions like: "What happens if we change this specific policy? Which other policies would also need to change?"

Evolvability of the policy provision function:

- *Limiting the impact of internal changes to other governance roles*: What is the extent to which local changes within the organisational boundaries of an actor who assumes the policy provider role trigger changes to other actors in the ecosystem? What happens when a policy is updated? What happens when the way in which policies are created is updated? Are changes contained locally? Is the method of policy definition independent from the method of governed resource description or the method of policy evaluation? Can changes to policies be effected without requiring re-engineering or disruption of operation for data providers or policy evaluators?
- *Limiting the impact of external changes to the policy provider role*: Conversely to the above, what is the extent to which changes within the organisational boundaries of data providers or policy evaluators trigger changes to the policy provider? What happens when the way in which governed resource descriptions are created, changes? What happens when the way in which policies are evaluated changes? Are changes contained locally? Can these changes be effected without requiring re-engineering or disruption of operation for the policy provider?

To illustrate this, consider the role of Cerebrate as policy provider in Scenario #5. The IT staff at Cerebrate has established a number of policies to govern a range of technical and business aspects in the cloud applications used by its departments. These policies need to be created and maintained internally at Cerebrate, but also communicated to external stakeholders who are required to enforce them, like Appregator –the cloud service broker.

How easy is it for Cerebrate to create policies on a variety of specialised areas such as confidentiality, privacy and safety of data? Do Cerebrate staff have access to a domain vocabulary they can use in formulating their policies? Can they build on previously created policies? Is there any tool to help users with identifying and correcting logical errors in the policies? Or with detecting conflicts with other policies created in the past?

What happens when Cerebrate introduces a new policy parameter related to the Service Level Agreements of the cloud applications that it uses (such as TalentForge)? Does this change require the policy evaluator (Appregator) to also make changes to its internal policy engine which evaluates SLA-related policies? Does the change require the resource provider (for instance, CloudDev, the platform on which TalentForge is deployed) to also make changes to how SLA-related data is extracted and communicated to the policy evaluator?

3.3.5 Data provider concerns

The goal of the data provider is analogous to that of the policy provider. Manageability and evolvability of the data provision process are the two main types of concerns.

Manageability of the data provision function:

- *Making it easy to create, maintain and share descriptions of governed resources (governance data)*: How easy is it for the data provider to aggregate data from different sources to produce homogeneous descriptions of governed resources? Does the data provider need to rebuild data extraction mechanisms from scratch per different source? Is there a generic and reusable mechanism for extracting and transforming data that can be configured and applied across different sources? How easy is it to allow consumption of those descriptions without catering to the specifics of each data consumer (i.e. policy evaluators)? Is the format chosen for governed resource description amenable to automated analysis? Is it possible to automatically validate resource descriptions and check for data quality issues? Can the data provider offer access to the data for distributed consumers and easily manage access permissions?
- *Making it easy to manage knowledge about resources, resource descriptions and the mechanisms that generate them*: How easy is it to maintain references from generated resource descriptions to the actual resources being described? Can the data provider keep track of metadata such as lineage, origin,

provenance information? Can the data provider easily track mappings between data sources and target transformations, or associations between data sources and the applicable extraction and transformation mechanisms? Can the data provider easily answer questions like: "What happens if we change this data source interface? Which are the extraction and transformation mechanisms that also need to change?"

Evolvability of the data provision function:

- *Limiting the impact of internal changes to other governance roles*: What is the extent to which local changes within the organisational boundaries of an actor who assumes the data provider role will trigger changes to other actors in the ecosystem? What happens when a data source is added? What happens when the structure or interface of a data source changes? What happens when the way in which data is extracted or aggregated from different sources changes? Are changes contained locally? Is the method of generating resource descriptions independent from the method of policy evaluation? Can changes to how governance data is produced be effected without requiring re-engineering or disruption of operation for policy providers or policy evaluators?
- *Limiting the impact of external changes to the data provider role*: Conversely to the above, what is the extent to which changes within the organisational boundaries of policy providers or policy evaluators trigger changes to the data provider? What happens when the way in which policies are evaluated changes, or when a new policy evaluation engine is added to the ecosystem? What happens when one or more policy providers change the way in which policies are defined? Are changes contained locally? Can these changes be effected without requiring re-engineering or disruption of operation for the data provider?

For instance, consider the role of CloudDev as Data Provider in Scenario #5. CloudDev offers an API through which one can obtain real-time data feeds about the activity, availability and updates of ISV applications hosted on CloudDev's platform. Cloud service brokers such as Appregator rely on this data to be able to offer their services to users of cloud applications, like Cerebrate –the HR agency.

A key concern of CloudDev in its role as data provider is to be able to manage the wide range and volume of data that it continuously tracks about different ISV applications on its platform in the best way possible. In parallel, CloudDev needs to be able to effectively and efficiently share this data with all of its partners –like Appregator. If necessary, CloudDev should be able to change how data is created and stored internally in its systems, but this change should not impair its ability to share data with external stakeholders.

How easy is it for CloudDev to aggregate data from different sources (inside or outside its own infrastructure) to produce descriptions of governed resources? Is there any

mechanism for extracting and transforming data that CloudDev can use across different sources? Does ClouDev need to cater to the specific needs of different data consumers such as Appregator or other cloud service brokers? Can CloudDev offer scalable and secure access to the data for distributed data consumers (policy evaluators) over the internet?

What happens when CloudDev adds a new data source, splits a data source in two, or changes how data is extracted from a source? What happens when CloudDev enhances the governed resource description with additional data? Do such changes trigger re-engineering or disruption of operation for policy evaluators such as Appregator or policy providers such as Cerebrate? What happens when Cerebrate introduces a new policy parameter related to the Service Level Agreements of the cloud applications that it uses (such as TalentForge)? Does this change require CloudDev to also make changes to how SLA-related data is extracted and communicated to the policy evaluator (Appregator)?

3.3.6 Policy evaluator concerns

The policy evaluator performs a function that depends on input from both policy providers and data providers. Like with policy providers and data providers one can see a comparable need for manageability and evolvability of the policy evaluation process. However, the specific concerns here are of different nature.

Manageability of the policy evaluation function:

- *Making it easy to evaluate policy conformance for different pairs of resource description and governance policy.* Does the policy evaluator need to implement policy evaluation modules/logic from scratch for every different pair or resource/policy? Is it possible to have policy evaluation logic in a generic and reusable form? Are policy representation formats and resource description formats common so as to ensure the policy provider can effortlessly understand them? Is the policy evaluation logic free from couplings to specific governance policies (or types of governance policies)? Is it free from couplings to specific governance resource description formats?
- *Making it easy to manage knowledge about relationships between policies and governed resources*: Can the policy evaluator easily keep track of which type of governed resource is a policy relevant to (so as to retrieve all relevant resources of that type from one or more data providers and validate them against that policy)? Can it easily track which policies apply to a specific resource (so as to retrieve the policies and evaluate them against the particular resource)?

Evolvability of the policy evaluation function:

- *Limiting the impact of internal changes to other governance roles*: What is the extent to which local changes within the organisational boundaries of an actor

who assumes the policy evaluator role trigger changes to other actors in the ecosystem? What happens when a policy evaluation engine is added or modified? Are changes contained locally? Is the method of evaluating policies independent from how resource descriptions are generated or policies represented? Can changes to how policies are evaluated be effected without requiring re-engineering or disruption of operation for policy providers or data providers?

- *Limiting the impact of external changes to the data provider role*: Conversely, what is the extent to which changes within the organisational boundaries of policy providers or data providers trigger changes to the policy evaluator? What happens when one or more policy providers change the way in which policies are defined? Or when the data provision process evolves? Are changes contained locally? Can these changes be effected without requiring re-engineering or disruption of operation for the policy evaluator?

To illustrate these concerns, let us take the example of Appregator who acts as policy evaluator in Scenario #5. Appregator is a cloud service broker offering continuous quality assurance and optimisation services to users of cloud applications. To offer its services, Appregator needs to be able to enforce a wide range of policies created by its customers like Cerebrate. It therefore needs to be able to obtain and understand these policies. Similarly, it needs to be able to obtain and understand all the data that it has to evaluate against these policies, like the data provided by CloudDev.

How easy is it for Appregator to support checking new customer policies against the same types of governed resources? Or to support checking new governed resources against the existing set of policies? Does new policy evaluation code need to be created, tested and maintained? Can Appregator easily provide an answer to a question like: "What are all the policies applicable to the type of that SLA description of TalentForge?" without being bound to the internal representation of the SLA resource at the data provider's side (the CloudDev platform)? Can Appregator easily answer the question "What are all the resources subject to governance by Cerebrate's policy for service uptime?" without being bound to the internal representation of the policy at the policy provider's (Cerebrate) side?

Can Appregator change the way it evaluates policies internally –for instance, by changing the process or the technology it employs without involving or affecting interoperability with any of its external stakeholders, like Cerebrate and CloudDev? What happens when Cerebrate introduces a new policy parameter related to the Service Level Agreements of the cloud applications that it uses? Does this change require Appregator to also make changes to how SLA-related policies are evaluated internally?

3.4 Implications on the design of governance support systems

Given the above-described characteristics of cloud service ecosystems where governance roles can be distributed between several collaborating actors, each of them carrying different viewpoints and concerns, what are the implications on how we design governance support systems? What are the design requirements shaped by such a collaborative environment? How can we provide that the system facilitates all stakeholders and their different sets of concerns, individually and as a whole?

3.4.1 Separation of concerns

A key insight to be derived from the analysis in the previous section is that, to facilitate governance in the context of a cloud service ecosystem, the design of the software system that is meant to support governance must achieve adequate separation of concerns between the roles of the policy provider, the governed resource data provider and the policy evaluator.

'Separation of concerns' is a design principle that involves decomposition of software according to one or more dimensions of concern, and has been at the core of software engineering for decades. The term was introduced by Dijkstra in 1974 [80] and initially referred to the concept of separating a software program into modules such that each module addresses a single concern, and can operate without requiring knowledge of how other modules operate.

Separation of concerns is closely related to the idea of 'information hiding' which was introduced by Parnas in 1972 [81] to address the problem of ever-growing size and complexity in software programs. Parnas suggested that programmers should segregate the design decisions in a computer program that are most likely to change, by encapsulating them in a module with a stable interface that exposes minimal information, so as to protect other parts of the program from extensive modification when the design decision is changed.

The benefit from applying this design principle is much improved system maintainability and reusability [82], [83]. Although initially conceptualised as an approach to improve modularisation of functionality within software programs, i.e. at the code-design level, the principle has since been applied at wider scope and architecture-design level to drive the decomposition of entire software systems according to both functional and non-functional requirements.

As defined by Mark Baker[39]: *"If we are attempting to separate concern A from concern B, then we are seeking a design that provides that variations in A do not induce or require a corresponding change in B (and usually, the converse). If we manage to*

[39] https://www.infoq.com/articles/separation-of-concerns

successfully separate those concerns, then we say that they are decoupled." The way in which separation of concerns facilitates software evolution is also highlighted by Mens and Wermelinger [82] who define *concern* as any criterion that allows us to decompose parts of the software that exhibit different rates of change or different types of change.

The observation by Parnas and Dijkstra that parts of a software system may exhibit different rates of change and different types of change holds true for any type of composite system, including systems of systems and software ecosystems. Therefore it also holds true for the different entities in a cloud service ecosystem that are engaged in a governance process. The entities assuming different roles in a governance process are bound to exhibit different rates of change and different types of change.

Applying the principle of separation of concerns to the design of a governance support system based on governance process roles would mean avoiding strong couplings between governance policies, governed resources, and policy evaluation engines. This would allow the actors assuming the relevant roles to operate collaboratively in the scope of the governance process, but also evolve their functions independently of each other.

3.4.2 Design requirements and quality attributes

In Table 1 below we summarise the design requirements that emerge from concerns associated with different governance roles and their functions. We present the target quality attributes (or "–ilities") that a governance support system will embody if it adequately addresses the sought separation of concerns.

Role	**Concern**	**Design requirement**	**Quality attributes**
Policy provider	Manageability of policy provision process	Abstract, domain-level policy representation language with support for automated analysis (checking self-coherence, contradictions/conflicts), and knowledge management (capturing policy dependencies/hierarchies).	Usability (policy comprehensibility), Maintainability (traceability, change scope minimisation), Testability (policy verifiability), Reusability (policy reusability), Interoperability, Availability (minimal disruption to deploy changes)
	Evolvability of policy provision process	Separation between provision of policy definitions and functions of other governance roles (provision of governance resource descriptions and policy evaluation). Standards-based interoperability for information exchange and decentralised coordination.	
Data provider	Manageability of data provision process	Structured resource description methods, with support for data validation and knowledge management. Data publishing mechanism with support for	Reusability (data extraction and transformation mechanisms),

		access control. Generic mechanisms for data extraction and transformation.	Maintainability (change scope minimisation), Testability (data validation), Interoperability, Availability (minimal disruption to deploy changes)
	Evolvability of data provision process	Separation between provision of governance resource descriptions and functions of other governance roles (policy definition and policy evaluation). Standards-based interoperability for information exchange and decentralised coordination.	
Policy evaluator	Manageability of policy evaluation process	Generic mechanism for governance policy evaluation. Support for knowledge management relating to policy evaluation (associations between policies and governed resources).	Reusability (policy evaluation logic), Maintainability (change scope minimisation), Interoperability, Availability (minimal disruption to deploy changes)
	Evolvability of policy evaluation process	Separation between policy evaluation and functions of other governance roles (provision of governance resource descriptions and provision of policy definitions). Standards-based interoperability for information exchange and decentralised coordination.	

Table 1. Governance support system design requirements and quality attributes

3.5 Summary

The aim of this chapter was to provide a basis for requirements thinking on designing governance support systems.

We started with presenting five exemplifying scenarios of governance in cloud service co-development ecosystems, each one demonstrating a different (and progressively more complex) setting where a governance support system is required. Going through the scenarios the reader will notice three recurring governance roles which interact in every governance process: the Policy Provider, the Data Provider and the Policy Evaluator. Entities in the ecosystem may simultaneously assume more than one of those governance process roles, while at the same time there can be more than one entity acting in the capacity of a certain governance role.

Associated with each role in the governance process is a distinct set of concerns and goals, which the respective stakeholder(s) would like to see satisfied. Some are role-level concerns which are local in scope and mainly relate to manageability of the process from the role's perspective. Others are ecosystem-level concerns which are global in scope, and mainly relate to evolvability and decentralisation of the process. Because of the fact that each role in a governance process has different concerns, the

entities that assume the relevant process roles are bound to exhibit different rates of change and different types of change over the lifetime of the governance process.

Based on the observations from the above analysis, we put forward the fundamental idea that underlies this research, which is that to facilitate governance in the context of a cloud service ecosystem, a governance support system must achieve adequate separation of concerns based on the roles of the policy provider, the governed resource data provider and the policy evaluator. By avoiding strong couplings between governance policies, governed resources, and policy evaluation engines, the relevant roles can operate collaboratively and evolve their processes independently of each other, at the same time.

The insights that underpin the idea of separating concerns when designing governance support systems, surfaced during the author's work in research projects CAST and Broker@Cloud. Both of these projects included an objective to create software systems supporting governance in cloud service ecosystems and presented the opportunity to analyse a variety of governance process scenarios. Analysis showed that any process of governance in the context of a software ecosystem is inherently a collaborative process where one can notice a recurring pattern of three fundamental roles which are continuously interacting.

At the close of the chapter we outline the design requirements shaped by such an environment and the software quality attributes (or "–ilities") that a governance support system will embody if it adequately addresses the sought separation of concerns.

In summary, the key takeaways from this chapter are the following:

1. Governance is effected through a collaborative process that may span multiple networked organisational units and enterprises.
2. The entities that participate in a governance process assume one or more of three fundamental roles: policy provider, data provider, or policy evaluator. A governance process may engage more than one entity in the same type of role (e.g. several different entities may act as policy providers in the same process).
3. Each governance process role is associated with a distinct set of concerns. These revolve around manageability and evolvability of the individual role's function (i.e. policy provision, data provision or policy evaluation).
4. Because of the different concerns associated with each governance process role, the entities who assume the relevant roles exhibit different rates of change and different types of change over the lifetime of the process.
5. In designing a software system to support collaborative governance processes, we need to ensure that the role-driven concerns of the different entities engaged in the process are independently addressed and simultaneously satisfied.

Chapter 4

A new foundation for governance support systems

4 A new foundation for governance support systems

4.1 Introduction

In the previous chapter we analysed the key requirements to be satisfied in designing governance support systems that are aimed at serving software ecosystems.

The questions that follow from this analysis are: Do state-of-the-art governance support frameworks/platforms represent a good fit to those requirements? Do their design assumptions acknowledge that there are three roles at play in the environment which are likely to be distributed among several actors in a network and that each role needs to be facilitated not only in isolation but in combination with the rest? If not, then what is the way in which governance support systems need to evolve? What would be a good basis to build on, to achieve this evolution?

This chapter begins with an analysis of how policy-based governance is supported in contemporary governance support systems. The analysis is based on examining the architecture and usage of two commercial governance technology platforms which are also open-source products. We walk through how such systems support the definition and enforcement of process and resource governance policies and how different entities are engaged in the governance process. We reflect on how well this design approach – as typified by the two platforms examined – corresponds to the requirements discussed in the previous chapter. We also identify the main change drivers to meet those requirements: enabling networked collaboration and improving the operational efficiency of governance support systems.

Motivated by the insight that semantic technologies can offer the sought evolutionary step in the design of governance support systems, which is based on the author's earlier experience with using semantic technology to enable enterprise interoperability, we introduce fundamental concepts from this field and discuss how open Semantic Web standards are relevant. On this basis, we put forward the thesis that Semantic Web technologies [14] and Linked Data principles [15] can provide the right foundation to develop governance support systems that satisfy the requirements of policy-based governance in the context of a cloud service ecosystem.

The chapter concludes with an overview of a conceptual framework which provides the fundamental architectural elements to be discussed in chapters 5 to 7.

4.2 Limitations of current governance support systems

As discussed in section 2.5, service governance technology in the form of registry and repository systems is finding its way into new applications for governance of cloud

service ecosystems. This has been shown to work well in environments where a single authority, the cloud platform provider, is the one that both sets the policies and ensures they are enforced.

However, some of the characteristics of contemporary registry and repository systems represent obstacles to the effectiveness and operational efficiency of governance in a cloud service ecosystem context, i.e. in a setting that is beyond the much simpler use case of governance in an isolated cloud application platform. A most fundamental limitation is found in the way in which registry and repository systems support the definition and enforcement of policies.

4.2.1 Definition and enforcement of policies

To understand how policies are defined and enforced in contemporary governance support systems, we analysed and compared two open-source commercial registry and repository products: MuleSoft Galaxy[40] and WSO2 Governance Registry[41]. The results indicate many similarities in the ways the two systems address policy definition and enforcement. The two products from MuleSoft and WSO2 are comparable in philosophy and functionality to many of their closed-source counterparts from vendors like IBM[42] or Oracle[43] and the approach adopted in these systems for policy definition and enforcement is representative of other governance technology platforms available in the market today [77].

In support of process governance, the registry and repository systems we analysed allow creating custom lifecycle definitions consisting of multiple phases, and associating them with different types of software entities (i.e. services or applications) or software artefacts (i.e. code files, configuration files, specification files, etc). Depending on the system, the lifecycle definitions can be created either through a straightforward point-and-click visual interface—which typically offers a limited set of options—or by means of XML specifications. For policies to be defined and enforced in relation to a lifecycle, new Java components must be coded against a special API that is provided for this purpose by each registry and repository system. The components must be packaged as JAR archives and deployed to the system's execution environment as extensions to the base functionality of the registry and repository system. The system, which in the cases of both MuleSoft Galaxy and WSO2 Governance Registry is implemented as a Java web application in a servlet container, needs to be restarted to complete the deployment of the extensions.

Resource governance is supported in a similar manner. For policies to be defined and enforced in relation to the structure and contents of different types of software artefacts stored on the registry and repository system, the operator of the system must code the

[40] https://www.mulesoft.com/resources/esb/application-service-governance
[41] http://wso2.com/products/governance-registry/
[42] http://www-03.ibm.com/software/products/en/wsrr
[43] http://www.oracle.com/technetwork/middleware/registry/

policies as extensions to the system's base functionality. This generally comprises two tasks.

The first task involves coding the conformance checking logic for each artefact type that needs to be validated. The validation is done either by reference to some external specification document (e.g. XML schema), or against a set of less explicitly defined rules that are coded directly in Java.

The second task involves coding extensions to trigger the execution of that conformance checking logic whenever an artefact of the particular type is added to the repository, or is modified. Depending on the system, this may be realised by implementing and registering event listeners, or by implementing combinations of filters and handlers that intercept whatever actions of interest are taking place in the system with regard to a given type of artefact.

In both tasks, the extensions are coded against special Java APIs provided by each registry and repository system for the particular purpose. Extensions need to once again be packaged as JAR files, deployed to a special directory, and the system must be restarted. Notably, this development and deployment cycle must be repeated every time an existing policy needs to be modified.

Section 8.4 in chapter 8 provides a closer look and specific examples of how WSO2 Governance Registry was extended based on the process outlined above to develop a governance support system for a cloud service ecosystem case study.

4.2.2 Impact on system quality attributes

The above described approaches to policy definition and enforcement by MuleSoft Galaxy and WSO2 Governance Registry present important limitations when required to support governance in the context of a collaborating ecosystem.

The absence of separation of concerns between the roles of the policy provider, data provider and policy evaluator is very prominent. Policy definition, data extraction and policy evaluation are entangled in the implementation of a single software component: a policy checking unit. The rules that a policy comprises are encoded in an imperative manner, directly in Java, as part of the same code that retrieves and validates the data. This violates the principle of separation of concerns as discussed in section 3.4 and can be shown to have many adverse effects.

Evolvability of the individual functions of different governance process roles

The most prominent adverse effect is hindered collaboration between distributed governance process participants. In governance scenarios like those examined in the previous chapter (where policies, data, and policy evaluation logic may be controlled and contributed by different entities), entangling policy definition, data extraction and

policy evaluation functions in a single software component makes collaboration impractical, if not impossible.

The fundamental assumption behind the design of contemporary governance technology platforms is centralised control of the governance process by a single entity. There is no provision for a governance process with "moving parts" which may exhibit different rates of change or different types of change while independently evolving. In short, contemporary design approaches typified by the platforms examined do not cater to the need that policy providers, data providers and policy evaluators have to ensure evolvability of their individual functions.

Manageability of the individual functions of different governance process roles

Beyond its limitations with respect to process evolvability needs, the approach taken by these governance support platforms is also limited in how it meets stakeholder needs to ensure manageability of their individual policy provision, data provision or policy evaluation functions.

To illustrate this, let us consider the challenge of manageability of the policy provision process, and start with focusing on policy traceability. Tracing the association of a policy at operational level to other policies representing strategic or tactical viewpoints within the same organisation is not possible. The transitive relationships that hold among policies at different levels of abstraction are opaque to users of the governance support system, as they are never made explicit.

Lack of traceability has a knock-on effect on policy maintainability. In the event that some high level organisational policy is revised, determining which low level operational policies are affected and should therefore be modified can be very difficult. Likewise, there is no way to know when some modification in a low level policy renders it incompliant to one or more higher level policies in the same organisation.

Another important limitation to manageability of the policy provision process is the lack of abstraction from the low level implementation details of a registry and repository system. Policies are coded against a low level Java API and deployed as extensions to the registry and repository system's core. The logic that underlies a policy is therefore represented at the same level of abstraction as the implementation logic of the registry and repository system itself.

This has implications with respect to usability and policy comprehensibility. It makes policy logic very difficult or even impossible to access and comprehend for users who may be experts in the domain but are not trained as software engineers. The same could be said, although possibly to a lesser degree, for engineers who are not familiar with a particular registry and repository system's design principles and extensibility mechanisms.

A related side effect is impaired policy verifiability. Logical inconsistencies or conflicts between policies, which could easily give rise to erratic system behaviour, become very difficult to detect. The logic that underlies a policy is hidden deep inside policy enforcement code, and is expressed in a form that is not amenable to automated processing and formal consistency checking.

Another affected aspect is policy interoperability. The low level of abstraction and tight coupling among policies and the registry and repository system is limiting the potential for portability of policies between different versions of a registry and repository system, or different systems altogether. Policy reusability is also prevented. If the registry and repository system is replaced or even significantly modified, the mechanisms for policy definition and enforcement need to be re-developed completely from scratch.

Also, quite importantly, deploying a new or modified policy to the registry and repository system requires taking it offline, since the system has to be restarted before new or modified extensions can take effect. In an always-on cloud computing setting which calls for uninterrupted operations, this disruption to availability is highly undesirable. Overall, governance agility is considerably restricted, because modifications are made impractical to apply on a frequent basis.

In addition to the limitations outlined above, which relate to manageability of the policy provision function, lack of separation of concerns also has an analogous adverse impact on manageability of the data provider and the policy evaluator functions. Maintainability, interoperability and reusability of data and policy evaluation logic are severely affected.

4.2.3 Dimensions of required enhancements

From the foregoing it becomes apparent that to address the problem of effective and efficient governance in the context of cloud service ecosystems, the methods by which policies are defined and enforced need to be drastically improved, along a number of dimensions.

Enabling networked collaboration

The primary need is enhancements to enable collaboration between the networked participants of the governance process. System design needs to be adapted so that it facilitates cross-departmental and inter-organisational collaboration between distributed ecosystem partners, through the online exchange of interoperable information that can be unambiguously interpreted by the software systems involved in the governance process.

Such enhancements to enable networked collaboration can be summarised as follows:

1. **Decoupling functions**: Allowing different ecosystem partners to work together on a governance process, while they retain the ability to evolve independently of each other, calls for decoupling the concerns associated with the different governance process functions (i.e. policy provision, data provision, and policy evaluation). The rules that comprise a policy and the data to be checked for compliance should be decoupled from the implementation of the policy evaluation logic. The three functions should be able to evolve independently of each other. It should be possible to modify the policy without necessarily having to modify the policy evaluation code. The same applies for couplings between the policy evaluation code and the data against which policies are evaluated. Changes to the representation of the data should not necessitate changes to the policy evaluation code.
2. **Ensuring interoperability**: Standards-based communication will allow distributed governance process participants to share process-related information between them. Policy definitions and data descriptions should be encoded in a way that allows them to be portable across systems.
3. **Increasing abstraction**: Allowing different ecosystem partners to work together makes it necessary to develop and use a common language. Communication needs to rise to a level that overcomes terminology differences between ecosystem partners. To achieve this, ecosystem partners can operate on a shared conceptual model that allows translating between their internal terms of reference and the terminology used at the ecosystem level. Concretely, defining the rules that a policy comprises should not require dealing with low level programming constructs of the governance support system. The policies should be possible to specify at a level of abstraction that is considerably higher than any implementation, and as close as possible to the constructs and concepts of the domain of interest. The same holds for the data to be checked against governance policies.
4. **Eliminating ambiguity**: Increasing the level of abstraction at which an idea is expressed opens up the problem of misinterpretation. This holds true for human communications and machine communications alike. Conveying the right context by reference to commonly accepted abstract terms must be combined with a means to exchange information that is structured and formal enough to allow unambiguous interpretation by machines.

Improving operational efficiency

A secondary need is enhancements to improve the operational efficiency of the governance process. In other words, enhancements to evolve the design of governance software so as to improve non-functional attributes of the governance process, such as reusability, maintainability, traceability and agility.

Such enhancements to increase operational efficiency can be summarised as follows:

1. **Maintainability**: The policy definition language should facilitate the verification of a policy's self-coherence as well as its consistency to other higher level policies to which it is associated. In addition, verification should reveal conflicts with other policies that would render the whole policy set in a registry and repository system incoherent. Verification should be possible to perform in an automated and reliable manner. Therefore, the policy definition language to be used should have formal underpinnings.
2. **Reusability**: Creating new policies and descriptions of governed resources should be possible to do by reusing existing policies and data. Creating a new policy or governed resource description should not necessarily require writing new code for policy evaluation. The policy enforcement mechanism should be as generic and reusable as possible.
3. **Traceability**: The way in which policies are defined should allow representing the associations they might have with other policies, such that relationships can be explicitly captured and dependencies can be traced. To make auditing easier, a common frame of reference should be employed for the definition of policies, in the form of a shared domain model, allowing higher-level policies to be linked to their lower-level manifestations.
4. **Agility**: It should be possible to modify policies and data sources in a seamless and agile manner, without needing to disrupt the operation of the governance support system for the changes to take effect.

4.3 The potential of semantic technology

The key insight that underlies the new approach presented in this dissertation is that the sought evolutionary step in the design of governance support systems can be achieved through the application of semantic technologies.

In his glossary of semantic technology terms[44] Michael Bergman offers a succinct, yet comprehensive definition for the broad concept represented by the term 'semantic technologies': *"a combination of software and semantic specifications that encodes meanings separately from data and content files and separately from application code. This approach enables machines as well as people to understand, share and reason with data and specifications separately."*

The insight that semantic technologies can provide the basis to enable networked collaboration while improving the operational efficiency of governance support systems is rooted in the author's past experience with researching solutions for enterprise interoperability [84]. Allowing heterogeneous and independent enterprise systems to participate in collaborative business processes calls for analogous changes

[44] http://www.mkbergman.com/1017/glossary-of-semantic-technology-terms/

in how systems interact – through standards-based communication, shared terminology and unambiguous interpretation. Applying semantic technologies to the challenge of enterprise interoperability has been shown to not only address these needs but to also allow improved maintainability, reusability, traceability and overall agility [115], [116].

Based on this experience, this research explored how knowledge representation and reasoning with open Semantic Web technology standards can be applied to advance the state of the art in governance technology platforms.

4.3.1 Logic-based knowledge representation and reasoning

Knowledge Representation and Reasoning (KR&R) is a field of Computer Science that emerged in the context of Artificial Intelligence (AI) and is one of AI's main subfields [85]. The roots of KR&R can be traced back to the late 1950s, when John McCarthy—the computer scientist who also introduced the concept of utility computing or cloud computing as it is known today—put forward a vision of intelligent systems that could exercise common sense [86].

In his seminal paper [87] McCarthy stated that a program could be said to have common sense if it was able to "*automatically deduce for itself a sufficiently wide class of immediate consequences of anything it already knows*". The premise for automated deduction of this kind would be to represent knowledge about a given domain in a way that enables the system to apply some generic rules of inference in order to draw conclusions or to take actions. McCarthy proposed to employ first order predicate logic as the basis for such a knowledge representation language.

As noted by Nebel [88] logic seems like a natural choice to address the problem of representing knowledge and reasoning about it, and indeed by the late 1980s logic-based methods had become prominent in the area of KR&R. A particular family of logic-based knowledge representation languages that has been increasingly attracting attention since that time is Description Logics (DLs) [89].

DLs have their roots in Brachman's structured inheritance networks [90], which were introduced to overcome some important limitations in the earlier KR approaches of semantic networks [91] and frames [92]; namely, the problems of ambiguity—due to their lack of formal semantics, and limited expressivity [89]. Structured inheritance networks were subsequently formalised as terminological systems, concept languages, and eventually as Description Logics [88]. What is common among DLs and their precursors is that, in contrast to earlier knowledge representation approaches, they are equipped with a formal, logic-based semantics [89].

Description Logics were so named because their focus is to facilitate the creation of descriptions for notions that are important in an application domain (also known as terminological knowledge). A domain is described in terms of concepts (sets of

objects), roles (binary relationships between objects), individuals (object instances), and their interrelationships. Descriptions of concepts, roles and individuals are given by formulating axioms in a variable-free syntax. A set of axioms constitutes a DL knowledge base.

Descriptions can be atomic—when single concepts or roles are defined, or complex—when composite descriptions are built from atomic ones using so-called concept and role constructors (e.g. concept or role union, intersection, complement, existential/universal restriction). Alternative combinations of concept and role constructors produce Description Logics that vary in terms of expressivity and complexity of inference. In most cases, however, the resulting DLs are decidable fragments of first-order predicate logic. A detailed account of the computational characteristics for several DLs is given in [89].

By convention, a DL knowledge base is regarded as consisting of two components: a tBox (terminological box), and an aBox (assertional box). The tBox contains the intensional knowledge that is available about the domain, i.e. axioms that describe properties of domain concepts and their interrelationships. Conversely, the aBox contains extensional knowledge about the domain, i.e. assertions about the relations holding among individuals, and among individuals and concepts (also called membership assertions). The distinction among tBox and aBox is in some sense similar to the distinction among database schema and actual data records in traditional relational databases. There exists, however, an important difference in the way asserted data is viewed in the context of databases and DL knowledge bases. While the information stored in a database is always regarded to be complete for the purposes of query answering, the information stored in an aBox is viewed as always being incomplete [89]. That is, records in a database are interpreted under a Closed World Assumption, while the assertions in an aBox are interpreted under an Open World Assumption when performing DL reasoning.

Once a DL knowledge base has been defined, it is possible to ask several kinds of questions about the described concepts and individuals, which can be answered by a DL reasoning engine. Typical reasoning tasks for tBox knowledge include concept classification (i.e. checking if a concept is subsumed by another), and concept satisfiability (i.e. checking that the description of a concept is coherent and not self-contradictory). Typical reasoning tasks for aBox knowledge are consistency checking (i.e. checking that the set of assertions in the aBox is consistent with the tBox) and individual classification (i.e. checking if a certain individual is an instance of a given concept).

The most important benefit that one obtains from representing knowledge about a domain with DLs is the ability to perform all those different types of reasoning (with well-defined properties in terms of soundness, completeness, decidability and complexity), and particularly, the ability to infer new implicit knowledge from what has been explicitly defined in a DL knowledge base.

The advantages that Description Logics provide as methods for formal knowledge representation and reasoning make them appealing for a variety of applications. They have been employed to create knowledge representation languages and systems for a great number of domains until today, such as natural language processing, medical informatics, product configuration, databases, and software engineering [89].

Notably however, the domain in which the application of DLs is considered to have been most successful is the development of ontology languages that provide a core building block for the architecture of the Semantic Web [86],[93].

4.3.2 Ontologies, Semantic Web standards and Linked Data

Ontology, as a term, has its origins in the field of Philosophy, in which it stands for the systematic study of what exists. In the context of Computer Science, however, ontology refers to an engineering artefact that is "constituted by a specific vocabulary used to describe a certain reality, plus a set of explicit assumptions regarding the intended meaning of the vocabulary words" [94]. In other words, an ontology is an "explicit specification of a conceptualisation" [95], providing a "shared and common understanding of a domain that can be communicated between people and heterogeneous and distributed systems" [96]. In the simplest case, an ontology may describe only a hierarchy of concepts, i.e. the subsumption relationships that hold between concepts in a domain. More fine-grained ontologies can also include other kinds of relationships between concepts, and specify constraints on their intended interpretation [94].

According to Tim Berners-Lee, the inventor of the Web, ontologies are a fundamental building block to realise the vision of the Semantic Web – an extension of the current Web in which "information is given well-defined meaning, better enabling computers and people to work in cooperation" [97]. Achieving this vision will transform the current Web from a massive repository of hyperlinked human-readable documents, into a global knowledge graph and an application platform.

In the view of Berners-Lee, in order to realise the Semantic Web vision, the data and services that are available on the Web must be augmented with semantic annotations that contain references to descriptive terms, with the meaning of such terms defined in ontologies [89]. The language in which these ontologies are to be encoded should allow for unambiguous interpretation by independently developed and autonomously operating software systems, and should facilitate reasoning upon the knowledge being specified in an automated and (computationally) practical manner. The formal, logic-based semantics with which Description Logics are equipped, the many years of experience of the DL research community in studying the computational properties of expressive DLs, as well as the implementation of highly optimised DL reasoning systems made the family of DLs an ideal starting point for that purpose [98].

Efforts in the area of creating an ontology language for the Semantic Web began in 2000, with the DAML project in the USA [99] and the On-To-Knowledge project in the EU [100]. Collaboration among the two teams led to the creation of the DAML+OIL language in 2001 [101], and soon later to the creation of OWL (Web Ontology Language) which was ratified as a standard by the World Wide Web Consortium (W3C) in 2004 [102]. In 2009, the OWL 2 recommendation [103] was introduced, providing some extensions and revisions to the initial specification while retaining backwards compatibility.

In both versions of OWL, the semantics of the language is definable via a translation into an expressive DL [104]. This correspondence allows tools and applications to exploit already known reasoning algorithms and reasoner implementations [89]. Semantically, OWL classes are the equivalent of DL concepts, OWL datatype properties and object properties correspond to DL unary and binary roles, and OWL instances are the equivalent of DL individuals. Examples of OWL-based ontology modelling will be introduced in detail in section 5.2.

The normative exchange syntax of the language is based on XML (Extensible Markup Language) [105] and RDF (Resource Description Framework) [106]—every OWL ontology can be represented as an RDF graph and can be serialised and exchanged as an RDF document. The use of URIs (Universal Resource Identifiers) [107] provides a way to globally and uniquely identify and reference the modelling constructs defined in an OWL ontology document (classes, properties and instances) across the Web. RDF Schema [108] provides part of the language's modelling vocabulary, and XML Schema [109] offers its datatypes to be used as concrete types in OWL.

This interaction of the OWL language with other W3C standards did not emerge unintentionally. It was a design objective in the context of creating a layered Semantic Web architecture, but also an attempt to leverage existing tool support and make the language as appealing as possible to existing user communities—especially the RDF community.

RDF is a standardised data model to create and share linked descriptions of resources accessible over the web. The RDF 1.0 specification was ratified by the W3C in 2004 (although existing as a W3C recommendation since 1999 in the early days of the Web), and RDF 1.1 was published in 2014. With RDF, resources are described by making statements about them in the form of subject–predicate–object expressions. These expressions are known as RDF triples. The RDF model for data representation will be introduced in more detail through examples in section 6.2.

Other technical specifications that are closely related to OWL and RDF and are often used in combination with OWL for developing Semantic Web applications are the rule languages SWRL [21] and RIF [110].

The SWRL (Semantic Web Rule Language) specification is a proposal for combining ontologies and rules which has not undergone a W3C standardisation process but is nevertheless endorsed quite widely and is supported by many tools. It allows extending the expressivity of OWL, particularly with respect to what can be said about properties of things in an OWL ontology (for instance, defining property chains), using a simple form of Horn-style rules [111]. Technically, it is a syntactic combination of the OWL language with the Datalog sublanguage of RuleML (Rule Markup Language).

RIF (Rule Interchange Format) provides a unified XML-based representation language for rules of different types, enabling them to be exchanged among heterogeneous rule systems over the Web. In doing so, it is also a rule language in its own right. RIF became a W3C recommendation in 2010. At its current state, the recommendation includes three dialects: the Basic Logic Dialect (BLD), the Production Rule Dialect (PLD), and RIF Core (the intersection of BLD and PRD). By means of an import mechanism, the rules defined in a RIF document can include references to elements defined in an RDF graph or in an OWL ontology available on the Web [112]. This allows rules to be formulated in terms of entities modelled in any ontology.

SPARQL (SPARQL Protocol And RDF Query Language) is a specification that first became a W3C recommendation in 2008. It defines a query language for data that is represented in the directed, labelled graph data format provided by the RDF standard. The SPARQL query language can be used to express queries across diverse data sources, whether the data is stored natively as RDF or is viewed as RDF via some kind of middleware [113]. By virtue of OWL's layering on top of RDF, this also applies to instance data defined in an OWL ontology. Query results are returned to the requestor as an XML document or an RDF graph, depending on the particular SPARQL query form that is being used (SELECT, CONSTRUCT, ASK, or DESCRIBE). The specification also defines a protocol for issuing queries to remote query processing services over the Web (via SOAP and HTTP bindings), and an XML document format for representing the results of SELECT and ASK queries. The SPARQL model for data querying will be introduced in more detail through examples in section 6.2.

The combination of URI, HTTP, RDF and SPARQL standards provide a toolset that allows developers to integrate heterogeneous data from distributed sources and share it between software applications with unprecedented efficiency. Tim Berners-Lee coined the term Linked Data to describe a set of best practices for online sharing of such structured data as interlinked datasets, using those basic web technologies [114].

Berners-Lee outlined four principles of Linked Data which provide a very simple guide for publishing data [114]:

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look up those names.

3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
4. Include links to other URIs, so that they can discover more things.

The first Linked Data principle advocates using URIs to uniquely identify not just web documents (as in the classic web), but any kind of abstract concept or concrete object. The second Linked Data principle advocates the use of URIs that can be dereferenced over the HTTP protocol and return a structured description of the identified object or concept. The third principle advocates the use of the Resource Description Framework (RDF), as a single data model for publishing structured and semantically unambiguous descriptions of objects and concepts, and the use of SPARQL as a common query language to allow exploring the RDF dataset and discovering relationships. The fourth principle stresses the use of hyperlinks between the things described by RDF to other descriptions of related things residing in different datasets [19]. However, there is one important note to how hyperlinking works. Whereas hyperlinks in the classic web are largely untyped, in a Linked Data context the hyperlinks that connect things between them have types, which describe the relationships between them. These types are defined in an ontology vocabulary.

These four rules have proven very effective in guiding data owners to publish Linked Data on the web, and the amount of data has grown rapidly. This data is often public as with the case of Linked Open Data, or it can be private, as is often the case with Linked Enterprise Data [115], [116]. In either case, the basic idea of Linked Data is to apply the general architecture of the World Wide Web to the task of sharing structured data on global scale [19], and bring interoperability to a whole new level.

4.3.3 Ontology-driven information systems engineering

The proliferation of the standards mentioned above, combined with the availability of several supporting tools have been a catalyst for the growing interest in recent years around the development of ontology-based software applications. But the potential benefits from employing ontology-based knowledge representation and reasoning to the development of software systems had been noticed and articulated much earlier.

The intersection of the fields of knowledge engineering and software engineering has been the subject of active study by a wide research community since as early as the mid-1980s [27]. Researchers in this community have been concerned with different ways in which knowledge representation and reasoning can improve the processes or artefacts of software engineering. In 1998 Guarino introduced the term ontology-driven information systems to describe the general class of software systems where the knowledge that is being represented and reasoned upon for the purposes of their development or operation has been formulated as an ontology [94].

According to Guarino [94], when considering the different ways in which ontologies can be used in the development of software systems, we can distinguish between two

orthogonal dimensions. The first is a temporal dimension, concerned with whether an ontology is used during the system's development time or run time. In the first case, the ontology is employed with the intention to affect the software process and not the software artefacts per se, so it is appropriate to speak of ontology-driven development. In the second case, where the ontology is intended to interact with the software artefacts themselves, we can further distinguish among ontology-aware and ontology-driven information systems. This distinction is a matter of whether the ontology is actually peripheral or central to the operation of the system. If the existence of an ontology is known only to a single component of the information system that uses it whenever needed, we speak of an ontology-aware information system. When the ontology is a central component of the system that continuously affects its behaviour, Guarino proposes to speak of a proper ontology-driven information system. The second dimension in Guarino's classification is a structural one, concerned with which components of the information system are being affected by the use of ontologies (presentation layer, logic layer, database layer), irrespective of when or how an ontology is used.

Uschold [16] makes a similar but simpler distinction between ontology-driven software engineering, where ontologies are used in the process of building an application but no ontology is used by the application itself, and ontology-driven information systems, where the ontology is additionally playing a significant role in the end application.

Happel and Seedorf [117] propose another classification that includes four general cases:

- Ontology-driven development, which involves the use of ontologies at development time for sharing descriptions of the problem domain,
- Ontology-enabled development, where ontologies are again used at development time but for actively supporting developers with their tasks,
- Ontology-based architectures, where an ontology is used as a primary artefact during run time, and
- Ontology-enabled architectures, where ontologies are used as auxiliary means for additional infrastructure support during run time.

Irrespective of which might be considered the most appropriate framework for classifying research in this area, the benefits that ontologies can bring in relation to information systems engineering are manifold.

Gasevic, Kaviani and Milanovic [118] provide a thorough analysis of the application of ontologies in different aspects of software engineering and the benefits that can be obtained. A similar analysis of applications and benefits of ontologies throughout the software lifecycle is given by Happel and Seedorf [117]. Bergman [119] points out that many of the benefits which are generally obtained by ontology-centric approaches to the development of information systems are attributed to the fact that the locus of effort

is shifted from software development and maintenance to the creation and modification of knowledge structures.

Uschold [16] cites six important benefits which result from the increased level of abstraction and the use of formal structures and methods in ontology-driven information systems:

- Reduced conceptual gap: developers interact with tools that are closer to their way of thinking about the problem domain rather than the implementation technology.
- Increased automation: formal structures are amenable to automated reasoning thus reducing human workload.
- Reduced development times: producing software artefacts that are closer to how we think, combined with reuse and automation, enables applications to be developed more quickly.
- Increased reliability: formal constructs, combined with increased automation, reduces the likelihood of human error.
- Increased agility/flexibility: ontology-driven information systems are more flexible, because changes can be made more easily and reliably in the model rather than in code.
- Decreased maintenance costs: increased reliability and automation reduces errors and formal links between models and code make the software easier to comprehend and thus easier to maintain.

4.4 Thesis statement

Software platforms which facilitate co-development relationships between different organisations foster the creation of environments best characterised as software ecosystems. Cloud service ecosystems represent one special class of software ecosystems (section 2.3). As these grow and become more complex, reliability is put at risk, requiring all stakeholders to exercise control over changes in the ecosystem that may affect them. This is a challenge of governance (section 2.4).

We put forward the view that a governance process within a cloud service ecosystem is inherently a decentralised, distributed and collaborative process spanning multiple organisational units and networked enterprises (section 3.3). Every governance process involves interaction between heterogeneous entities which perform one or more of the following functions: providing the policies, providing the data to be evaluated against policies or carrying out the actual policy evaluation. The concerns associated with each of the three functions are very different and cause the entities that assume these roles to

exhibit different rates of change and different types of change over the lifetime of the governance process.

Therefore, in designing a software system to support collaborative governance processes, we need to ensure not only interoperability, which represents an obvious challenge, but need to also ensure that the role-driven concerns of the different entities engaged in the process are independently addressed and simultaneously satisfied. We need governance support systems that achieve adequate separation of concerns between the roles of the policy provider, the governed resource data provider and the policy evaluator (section 3.4). By avoiding strong couplings between governance policies, governed resources, and policy evaluation engines, the relevant stakeholders can operate collaboratively and evolve independently of each other at the same time.

My thesis is that governance support systems that satisfy these requirements are both feasible and useful to develop through a framework that integrates Semantic Web standards and Linked Data principles.

Semantic Web standards and Linked Data principles were designed to support the exchange of large volumes of heterogeneous and continuously evolving data, in a way that allows machines to unambiguously interpret the meaning of the data from each source, link together data from multiple different sources and independently generate new knowledge, without requiring any upfront investment in system-to-system integration (section 4.3).

Heterogeneity, distribution and continuous evolution are the fundamental characteristics of the web. Semantic Web standards and Linked Data principles have been designed on that foundation. The key insight that underlies this thesis is that these same characteristics are also fundamental properties of governance processes in cloud service ecosystems. The challenge in designing a software system architecture to support governance in a cloud service ecosystem is a challenge of coping with heterogeneity, distribution and continuous evolution.

As an additional benefit, beyond the capability to enable networked collaboration, Semantic Web standards can also guarantee the higher level of operational efficiency that ecosystem governance processes require. By virtue of the formal semantics, modelling abstraction and standards-based interfacing embodied by the standards, the design of governance support systems can benefit from improved reusability, maintainability, traceability and agility.

4.5 PROBE framework

The above described benefits of ontology-centric approaches to information systems engineering, in combination with the Semantic Web standards and Linked Data

technologies currently available, provide a promising foundation for a new approach to designing and implementing governance support systems for cloud service ecosystems.

We here propose a conceptual architecture framework for developing governance support systems which we will refer to as the PROBE framework (policy-driven governance in cloud service ecosystems).

The approach we put forward comprises four core components:

- First, a shared governance ontology to provide the basic vocabulary and modelling constructs for describing both (i) the governance policies and (ii) the governed software resources made available in the ecosystem.
- Second, ontology-based definitions of governance policies based on a uniform logic-based encoding method, referencing the shared governance ontology model.
- Third, mechanisms to generate abstract, semantic descriptions of the different kinds of governed software resources, by means of transformation from their native representation into Linked Data with references to the shared governance ontology.
- Fourth, a generic and reusable policy evaluation engine to check if the descriptions of governed resources conform to the relevant governance policies.

Figure 8 illustrates the PROBE framework architecture in conceptual form.

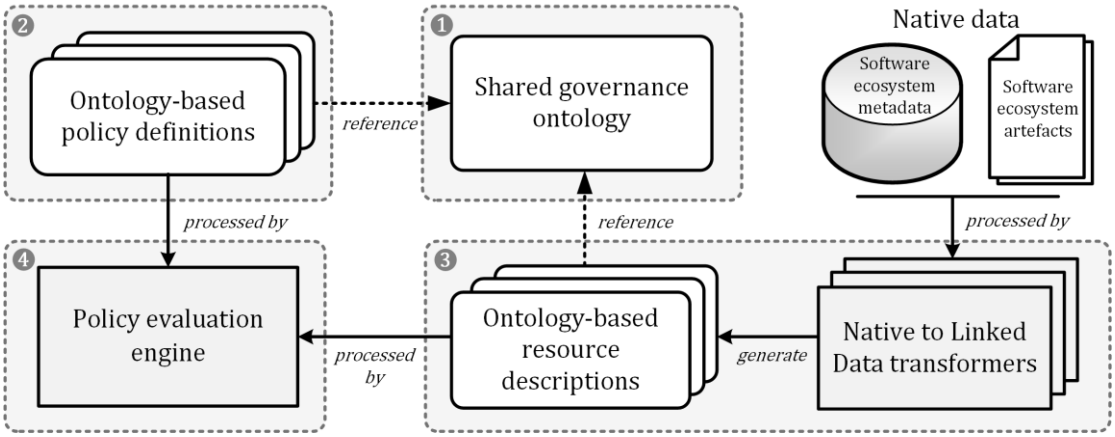

Figure 8. Overview of PROBE framework architecture for governance support systems

Chapters 5, 6 and 7 explain the purpose served by the different components of this conceptual framework and show how each one can be realised with the application of contemporary web technology standards and tools.

4.6 Summary

This chapter opens with an analysis of how policy-based governance is supported in contemporary governance technology platforms, based on a study of two widely used products that offer the opportunity to be studied in depth, as they are distributed under a commercial open source license. We walk through how these systems support the definition and enforcement of process- and resource-related policies and reflect on how the general approach typified by those systems meets the design objectives discussed in the previous chapter.

The previous chapter introduced the idea that systems aimed at supporting governance processes in a cloud service ecosystem are required to simultaneously facilitate the concerns associated with three distinct roles in the governance process: the policy provider, the governed resource data provider and the policy evaluator. Separating the concerns associated with each role is the only way to facilitate all three roles at the same time, allowing the entities that take on these roles to manage and evolve their internal functions in an effective and efficient way.

Our analysis in this chapter shows that contemporary governance technology platforms cater to much less complex governance requirements; networked collaboration is a requirement that they are simply not designed to meet. The fundamental assumption behind the design of contemporary governance technology platforms is centralised control of the governance process by a single entity. There is no provision for a governance process where multiple collaborating entities may exhibit different rates of change or different types of change while they independently evolve.

Consequently, there is no architectural provision for separation of concerns between the roles of the policy provider, data provider and policy evaluator. Policy definition, data extraction and policy evaluation are entangled in the implementation of a single software component: a policy checking unit. In more concrete terms, the rules that a policy comprises are encoded in an imperative manner, for instance, directly coded in Java, as part of the same code that retrieves and validates the data.

Based on this observation we discuss the dimensions of required enhancements that we need to bring to contemporary governance technology platforms so that we can cater to the advanced needs of cloud service ecosystems. We identify several aspects in which this technology needs to evolve, centred on two objectives: enabling networked collaboration and increasing operational efficiency in governance support systems.

This analysis of the problem space leads into introducing the wider solution space. That is, architectural principles and concrete implementation technologies from the domain of semantic technologies, such as ontology-based knowledge representation and reasoning, ontology-driven information systems engineering and Semantic Web standards.

The insight that semantic technologies can provide the basis to enable networked collaboration while improving the operational efficiency of governance support systems is rooted in the author's past experience with researching solutions for enterprise interoperability. The challenge of enabling heterogeneous enterprise systems to participate in collaborative business processes bears many similarities to the challenge of enabling ecosystem-wide governance processes. Experiences from applying knowledge representation and reasoning with open Semantic Web standards to the former domain can therefore readily be transferred to the latter.

To explain the background and motivation for this approach we introduce some fundamental concepts from the domains of logic-based knowledge representation and reasoning, ontologies for the Semantic Web and related standards, and ontology-driven information systems engineering. This leads to presenting the thesis supported by this research, i.e. that a software architecture satisfying the advanced requirements of governance in cloud service ecosystems is both feasible and useful to realise on the basis of Linked Data principles and Semantic Web standards.

On this basis we introduce a conceptual framework architecture for governance support systems that realises this approach. The fundamental architectural elements of the PROBE framework are discussed in the three chapters that follow (chapters 5, 6 and 7), which aim to demonstrate that a governance support system solution based on this framework is indeed feasible to create.

The key takeaways from this chapter can be summarised as follows:

1. State-of-the-art governance technology platforms are designed to meet requirements that are much simpler compared to those of governance support systems aimed at serving cloud service ecosystems. By virtue of its nature as a distributed and collaborative process, supporting governance in this new context requires systems to enable networked collaboration and to guarantee a higher level of operational efficiency.
2. The thesis supported by the research presented in this dissertation is that a software architecture satisfying the advanced requirements of governance in cloud service ecosystems is both feasible and useful to realise on the basis of Linked Data principles and Semantic Web standards.
3. The insight that underlies this thesis is that semantic technologies of this kind have already been shown to provide successful solutions in related problem domains such as inter-enterprise interoperability and policy engineering, and there are lessons learnt which can be readily transferred.
4. The domains of ontology-based knowledge representation and reasoning, ontology-driven information systems engineering and Semantic Web research can provide architectural principles and concrete implementation technologies.

5. Our proposed framework architecture for policy-driven governance in cloud service ecosystems (PROBE) comprises four core components: a shared governance ontology; a repository of ontology-based policy definitions; mechanisms to generate ontology-based resource descriptions; a governance policy evaluation engine.

Chapter 5

Defining governance policies

5 Defining governance policies

5.1 Introduction

In previous chapters we went through the fundamentals of cloud service ecosystems, we analysed the key requirements associated with developing governance support systems suitable for cloud service ecosystems, we discussed the potential of semantic technologies as a new foundation for the development of such systems and we presented an abstract software framework architecture that builds on this foundation.

Our proposed framework architecture comprises four core components:

- a shared governance ontology;
- a repository of ontology-based policy definitions;
- mechanisms to generate ontology-based resource descriptions;
- a governance policy evaluation engine.

In this chapter we discuss an instantiation of the first two of these components with contemporary web standards. We describe the conceptualisation and representation of an ontology which serves as a shared vocabulary to define governance policies, but also provides the vocabulary to describe governed software resources. In this context we also propose some governance policy modelling patterns with the help of real-life examples drawn from the governance policy dataset of an actual cloud service ecosystem. Lastly, we also discuss related work on ontology-based policy representation and related semantic technologies.

5.1.1 Governance from the policy provider's perspective

As discussed in chapter 3, the policy provider role is responsible for creating, maintaining, and providing the governance policies that need to be enforced. A single policy may involve more than one governed entities in a cloud service ecosystem, whereas a resource may be governed by more than one policy originating from different policy providers.

The goal of the policy provider is to be able to effectively and efficiently manage policies internally, and to communicate these policies to other partners who need to comply or enforce them.

- Internal management objectives: To be able to freely modify existing policies while containing changes locally, i.e. without necessitating any corresponding changes to third-parties such as data providers or policy evaluators. To be able to easily create new or modify existing policies which may require traceability

of logical dependencies between policies, detection of contradictions with other policies and debugging of complex policy logic.

- External communication objectives: To be able to exchange/share policies in a platform-agnostic way, without needing to consider how the policies will be later processed and evaluated, or how data concerning the governed resources is represented internally by data providers. To be able to efficiently and quickly have new or modified policies enacted/deployed to the destinations where they need to be enforced.

5.1.2 Governance policies from the ecosystem's perspective

Observed from an ecosystem-wide perspective, policies exhibit the following three characteristics.

Heterogeneity:

- Governance policies concern very different aspects of governance objectives from the strategy level down to the operational level, and many different characteristics of a software ecosystem resource, from the pricing model or localisation details of a software unit, to its lifecycle stage.
- Policies are initially represented in very dissimilar native formats (documents, web pages, structured files).

Physical distribution:

- Governance policies are stored in different locations. Software ecosystem partners are distributed, and so are the policies that ecosystem partners may wish to enforce on governed ecosystem resources.
- Policies need to be exchanged over the internet between ecosystem partners.

Fragmented ownership and control:

- Governance policies are owned by multiple independent partners and may evolve (modified, removed) independently of other governance process components (data or policy evaluation engines).
- Policies are owned by independent partners who are free to choose and use their own terms of reference in their local/native policy definitions.

5.2 Governance ontology

5.2.1 Basic characteristics

As discussed in section 4.3.2, an ontology represents a "shared and common understanding of a domain that can be communicated between people and heterogeneous and distributed systems" [96].

Ontology development starts with determining scope and intended usage of the ontology model; i.e. what should the domain cover and what will the ontology artefact be used for. Scope and intended usage dictate the appropriate level of abstraction. At the highest level of abstraction we have foundational ontologies (also referred to as upper or top-level ontologies) such as GFO or DOLCE, which aim to formalise very general concepts that are common and reusable across different knowledge domains [120]. At the lowest level of domain abstraction we have application-specific ontologies which can be extremely narrow in focus. In between those two ends of the spectrum there is a wide variety of domain ontologies which may formalise a mix of high-level and low-level domain concepts in a common ontology – or in a set of interlinked ontologies.

Beyond domain abstraction, other important aspects are the extent of formalisation in the ontology – i.e. the degree of axiomatisation in the description of classes, attributes and relations of domain concepts, and the size of the ontology – i.e. the number of domain entities described in the ontology artefact.

In the context of this research our primary focus wasn't to develop a foundational domain ontology for cloud service ecosystems governance, although the ontology presented here could be extended to provide one. Our goal in designing this ontology was to validate the feasibility of the framework presented in chapter 4. As such, the goal was to develop an ontology closer to the application-specific end of the spectrum, featuring relatively extensive axiomatisation over an adequate number of constructs.

The governance ontology was developed based on the governance support system requirements that we analysed in the scope of research project CAST [22], [121],[122]. The constructs described in the ontology mirror the exact structure and characteristics of an actual ecosystem-oriented PaaS system, the CAST platform. In fact, the description of example scenario #3 which was provided in section 3.2.3 above was inspired by the characteristics and governance requirements of the CAST cloud service ecosystem. Through the process of requirements analysis for the CAST governance support system we derived a set of 37 governance policies of varying scope and complexity.

For the remainder of this chapter and the chapters to follow we will be using governance policy examples based on the CAST project case study, which is introduced in full detail in chapter 8.

The resulting governance ontology comprises:

1. A small set of core domain concepts which are generic and independent of any concrete cloud service ecosystem governance requirements, such as the notions of cloud platform resources, resource collections, lifecycle states and state transitions.
2. Modelling constructs specific to the governance requirements of CAST. These correspond to the different types of logical entities found on the CAST platform such as different kinds of software units (solutions, apps and services), different kinds of software artefacts (e.g. deployment descriptors, interface definitions, pricing specifications, localisation files, images), different lifecycle states (development, testing, review, beta, production, deprecation, end-of-life), etc.
3. The definitions of policies governing CAST platform resources, based on the set of the 37 governance policies produced from project CAST.

For reasons of standards-based interoperability and tool support, the language we have adopted for the specification of the governance ontology is OWL 2 [103].

The full ontology model comprising the platform-independent governance concepts, the platform specific governance concepts and the governance policies has the following characteristics:

- Size: 170 classes, 30 properties, 29 individuals (constants), 7 SWRL rules
- DL expressivity: $\mathcal{ALCROIQ}^{(\mathcal{D})}$[45] plus DL-safe [123] SWRL rules
- Language features specific to OWL 2: XSD facets, Keys

In the rest of this section we will provide a brief overview of the different types of domain/policy modelling constructs with the help of examples. The full specification of the governance ontology can be found at www.ecosystem-governance.com.

5.2.2 Class hierarchy

The ontology class hierarchy specifies the subsumption relationships that hold between the 170 concepts in the domain being modelled.

An example of such a subsumption relationship is illustrated in Figure 9.

[45] For an overview of DL reasoning characteristics and the notation to describe the modelling construct composition of DL languages refer to http://www.cs.man.ac.uk/~ezolin/dl/

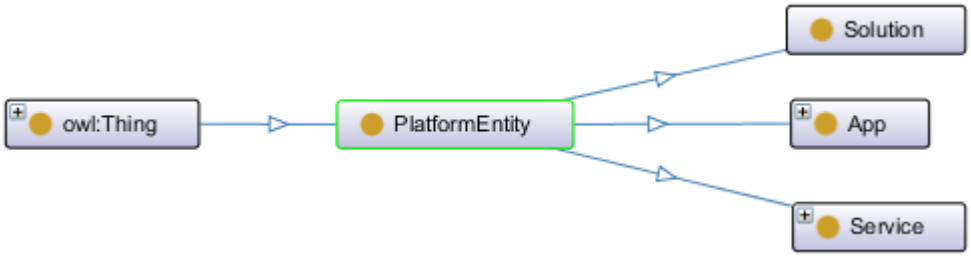

Figure 9. Excerpt from governance ontology class hierarchy

The concept of `PlatformEntity` refers to the basic software unit on the CAST platform. Software developed against the CAST platform can either be a `Service` (i.e. a web API), an `App` (i.e. a UI-complete application), or a `Solution` (i.e. a bundle of interoperable apps).

Table 2 provides the axiomatic description of the `PlatformEntity` class in OWL 2, using Manchester Systax[46].

```
Class: PlatformEntity

    DisjointUnionOf:
        Solution, App, Service
```

Table 2. Description of PlatformEntity class (OWL Manchester Syntax)

`PlatformEntity` is defined as the disjoint union of `Solution`, `App` and `Service`, while those three concepts are in turn defined as subclasses of `PlatformEntity` in their own class expressions (not shown in the code excerpt above).

A disjoint union class expression in OWL 2 allows one to define a class as the union of other classes, all of which are pairwise disjoint. Intuitively, this means that no software unit on the CAST platform can ever be both a `Solution` and an `App` at the same time, for instance. `PlatformEntity` is a root class in our governance ontology, which is why it is not defined as a subclass of any other class, except for `owl:Thing`, the top class in every ontology according to the semantics of OWL 2. Every individual in the OWL world is a member of the class `owl:Thing`. Consequently, every user-defined class is implicitly a subclass of `owl:Thing`.

[46] For the rest of the excerpts of OWL 2 DL code provided in this dissertation we will be using Manchester Syntax [103] in place of the equally formal but much less readable mathematic Description Logic notation.

5.2.3 Object and data properties

The governance ontology includes 14 object properties and 16 data properties. Object properties are used to represent relationships between individuals (instances) of classes, while data properties connect individuals with RDF literals or simple types defined in accordance with XML Schema datatypes (such as xsd:integer and xsd:string).

Table 3 provides an example object property specification excerpted from the ontology. The `hasDependency` property can be used to specify that some `PlatformEntity` depends on another `PlatformEntity`, such as when a `Solution` depends on some `App`, or when an `App` depends on a specific `Service`.

The property is defined as the inverse of `isDependencyOf`. The property is irreflexive — that is, no individual can be connected to itself through this property. It is also defined as assymetric, meaning that, if an individual x is connected along this property to an individual y, then y cannot be connected along the same property to x. In this domain this prevents circular dependencies between software units on the cloud platform.

```
ObjectProperty: hasDependency

    Characteristics:
        Irreflexive,
        Asymmetric

    InverseOf:
        isDependencyOf
```

Table 3. Description of hasDependency object property

Table 4 provides an example data property specification. The `hasSizeInKB` property can be used to specify the size of any software artefact residing on the cloud platform, such as the size of deployment archives or the size of iconography and screenshots bundled with the description of a software unit. The property is restricted to range over `xsd:decimal` values and is defined as functional, meaning that an individual x cannot have more than one connection to literals along this property (in other words, it cannot have more than one value for file size).

```
DataProperty: hasSizeInKB

    Characteristics:
        Functional

    Range:
        xsd:decimal
```

Table 4. Description of hasSizeInKB data property

5.2.4 Individuals

In Description Logics, a distinction is drawn between the tBox (terminological box) and the aBox (assertional box). When creating a DL knowledge base using OWL 2 as modelling language, class expressions, object properties and data properties belong to the tBox of the knowledge base. OWL 2 individuals (instances) belong to the aBox.

The 29 individuals included in the governance ontology do not represent instances of the class expressions mentioned earlier. Their role is to assist in domain modelling by providing some fundamental classification constants for governed resources. These classification constants are used inside class expressions wherever required.

Table 5 provides an example individual specification. It defines a constant named `_SOAPService` which is defined as pairwise disjoint from another individual representing the constant `_RESTService`.

```
Individual: _SOAPService

    Types:
        ServiceInterfaceClassification

    DifferentFrom:
        _RESTService
```

Table 5. Description of _SOAPService individual

OWL 2 doesn't adopt a Unique Name Assumption (UNA) and it is sometimes necessary to be able to specify when two individuals are not the same object, to support inferencing and enable detection of contradictions in the knowledge base. This is achieved using the `owl:DifferentFrom` language construct.

In practice, the `_SOAPService` individual is used as a classification constant to distinguish between different types of services delivered by the CAST platform with respect to their interface type. Some services may be implemented such that they can be invoked through the SOAP protocol and others through a REST protocol. Any service delivered by the CAST platform can either expose a SOAP or a REST interface, not both at the same time.

Table 6 presents the class expression of `SOAPService` which makes use of the individual `_SOAPService` as a classification constant.

```
Class: SOAPService

    EquivalentTo:
        Service
         and (hasInterfaceClassification value _SOAPService)
```

```
    SubClassOf:
        Service
```

Table 6. Description of SOAPService class

5.2.5 SWRL rules

The governance ontology is making use of the full expressiveness capabilities of the OWL 2 language. The Description Logics dialect corresponding to the developed governance ontology is $\mathcal{ALCROIQ}^{(D)}$, which suggests there is a high degree of axiomatisation in the policy descriptions. Nevertheless, there are certain cases of policies from the CAST project dataset that cannot be expressed with OWL 2 axioms.

OWL 2 DL is a decidable fragment of first order predicate logic, but there are cases where class membership conditions or property relationship conditions cannot be directly represented in OWL 2 [124].

For instance, a first order predicate logic rule such as

```
Solution (s)
∧ hasDependency (s, a1)
∧ hasDependency (s, a2)
∧ conflictsWith (a1, a2)
→ ConflictingDependencySolution (s)
```

which defines the class `ConflictingDependencySolution` as consisting of CAST platform solutions whose dependencies are conflicting, is not expressible in OWL 2.

The expressivity of an OWL 2 ontology can however be extended to support such domain/policy modelling requirements by adding SWRL (Semantic Web Rule Language) rules to the ontology. SWRL rules are Datalog rules with unary predicates for describing classes, binary predicates for describing properties, and some additional n-ary predicates for language built-ins.

Table 7 presents the equivalent SWRL rule expression of the abovementioned class membership conditions rule.

```
(Class: ConflictingDependencySolution)

    Rules:
        Solution (?s),
        hasDependency(?s, ?a1),
        hasDependency(?s, ?a2),
        conflictsWith(?a1, ?a2)
            -> ConflictingDependencySolution (?s)
```

Table 7. Description of ConflictingDependencySolution class membership conditions via a SWRL rule

5.2.6 Advanced OWL 2 features

In addition to the above, the governance ontology makes use of some relatively advanced language capabilities which were added to the OWL specification with the introduction of OWL 2.

The most significant ones are Facets and Keys [125].

- Facets: OWL 2 offers a wider set of datatypes compared to OWL (v1) and supports restrictions of data values by facets, as in XML Schema. Those restrictions make it possible to specify acceptable datatype values via constraining facets which restrict the range of values allowed. For text, `xsd:minLength` and `xsd:maxLength` can be used to restrict string length. For numbers, `xsd:minInclusive` and `xsd:maxInclusive` can be used to restrict values.
- Keys: Keys offer increased expressive power as they enable an OWL-DL reasoning engine such as Pellet [127] or Hermit [128] to uniquely identify individuals of a given class by values of key properties. The OWL 2 construct `owl:hasKey` allows keys to be defined for a given class. An `owl:hasKey` axiom states the (data or object) property through which it is possible to uniquely identify each named instance of the class. If two named instances of the class have the same value for each of key properties, then these two individuals are inferred to be one and the same.

5.3 Method for creating governance policies

5.3.1 Process and resource governance

In section 2.4.1 we introduced a definition of cloud service ecosystem governance that draws a conceptual distinction between governing ecosystem resources and governing ecosystem processes. Our analysis to date supports that any form of policy facilitating cloud service ecosystem governance can be abstracted to the level of either resource or process governance.

One of the outcomes from the work carried out by the author in the scope of research project CAST was the observation that, in the context of a cloud service ecosystem which is created around a PaaS platform, process governance is effectively mapped onto lifecycle management, whereas resource governance is mapped onto artefact validation.

5.3.1.1 Policies for lifecycle management

Process governance via lifecycle management policies is concerned with ensuring a structured and disciplined approach to introducing software units developed by different ecosystem partners onto the deployment and execution environments, modifying them, or removing them. Central to this is the notion of a lifecycle model defining the phases that every different managed software unit needs to proceed through, as well as the preconditions associated with the transition from one lifecycle phase to the next.

For instance, one of the lifecycle governance policies defined for the CAST platform states that a precondition for allowing an app to proceed from the review phase to the beta testing phase, is for the app to be associated with a quality review report that contains a positive evaluation. In addition to this precondition, the app must continue to satisfy all preconditions defined for previous transitions (i.e. the transition from local development to sandboxed testing and from sandboxed testing to review).

5.3.1.2 Policies for artefact validation

Resource governance via artefact validation policies is concerned with ensuring that all artefacts and metadata associated with software units of cloud services are conformant to technical, business or legal constraints which are defined by the platform provider or by different ecosystem partners. Central to this concept is the notion of artefact specifications which place constraints on the valid structure and contents that different kinds of configuration, specification or code artefacts are allowed to have.

For example, one of the artefact validation policies defined for the CAST platform states that the interface specification (Web Services Description Language or WSDL) of every external web service used by one or more apps, must contain exactly two non-identical endpoint URLs. These URLs must point to different servers on which the service is deployed (primary and backup endpoints). The rationale of this policy is to provide a failover alternative in case the primary server that hosts the service becomes unavailable.

5.3.2 Policy encoding patterns

The method we present here introduces the approach of expressing lifecycle management and artefact validation policies as OWL classes. This is done by creating a new OWL class for every policy to be defined and constructing equivalent class axiom expressions. This section provides several examples that illustrate how this can be achieved.

We put forward the idea of a policy encoding pattern whereby process and resource governance policies can be formulated in either positive or negative form. A related

concept was applied in the representation of security policies with KaoS [18] and Rei [126], which are discussed in section 5.4 of this chapter.

In some cases, it is much more intuitive to express policy constraints in terms of what should necessarily hold in the domain (positive form), rather than what should not be the case (negative form). In other cases it can be the opposite. And there are always cases where the only possible way to express the constraints imposed by a policy is in one of the two forms, not the other.

The logic-based policy representation foundation in our approach provides the flexibility to define a policy in any of the two forms, depending on what is best, while affording powerful reasoning capabilities for the purposes of policy evaluation.

5.3.2.1 Positive formulation for artefact validation policies

The purpose of artefact validation policies is to regulate the structure and contents of software-related resources of a cloud service ecosystem. Similarly to how integrity constraints help ensure accuracy and consistency of data in databases, constraints on cloud service ecosystem resources will ensure conformance to governance requirements. The policy evaluation process will reveal whether or not an artefact is valid – the outcome of the process is always Boolean.

A positive formulation of a resource governance policy describes the resource in terms of the conditions that make it a valid artefact. Conversely, a policy expressed in negative form describes the resource in terms of the conditions that make it an invalid artefact.

To illustrate the policy encoding process let us consider a CAST platform app artefact as a concrete example; the case of `AppScreenshot`. As suggested by its name `AppScreenshot` is a resource that provides a screenshot of an `App` deployed to the CAST platform. This resource will be used as part of the `App` description to be placed in the cloud platform's app store/directory.

Policy encoding for `AppScreenshot` will proceed as follows:

1. Step 1: Create a *primitive* OWL class[47] `ValidAppScreenshot` which is asserted as a subclass of `AppScreenshot` and `ValidAppArtefact`. Should the

[47] In Description Logics languages like OWL concepts can be either *primitive* or *defined*. OWL classes that are only described in terms of necessary conditions (i.e. by asserting their superclasses) are known as primitive classes, whereas classes described in terms of both necessary and sufficient conditions (i.e. by asserting an equivalent class axiom) are known as defined classes. With defined classes, a reasoner can deduce that any individual that satisfies the definition will belong to the class. With primitive classes a reasoner will not draw this conclusion.

governance policy be encoded in positive form, this class will be later converted to a *defined* class in step 4.

2. Step 2: Create a primitive OWL class `InvalidAppScreenshot`, which is also asserted as a subclass of `AppScreenshot` and `InvalidAppArtefact`. Should the governance policy be encoded in negative form instead of positive, it will be this class that will later be converted to a *defined* class.
3. Step 3: Create/modify the description of superclass `AppScreenshot`, asserting it is a subclass of `AppArtefact` and a disjoint union of the newly created `ValidAppScreenshot` and `InvalidAppScreenshot`.
4. Step 4: Depending on whether a positive or negative formulation is more convenient, construct an equivalent class axiom expressing the policy conditions and attach it to `ValidAppScreenshot` or to `InvalidAppScreenshot`, respectively, converting one of the two into a defined OWL class.

Figure 10 illustrates the subsumption relationships defined between classes to facilitate resource governance for `AppScreenshot` artefacts.

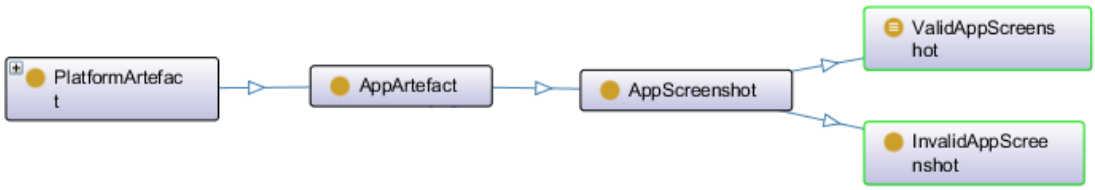

Figure 10. Class hierarchy in policy encoding pattern for the validation of AppScreenshot resources

`AppScreenshot` is an `AppArtefact`, which is itself a `PlatformArtefact`. An `AppScreenshot` can either be a `ValidAppScreenshot` or an `InvalidAppScreenshot`. The OWL class description for `AppScreenshot` is provided in Table 8.

```
Class: AppScreenshot

    DisjointUnionOf:
        ValidAppScreenshot, InvalidAppScreenshot

    SubClassOf:
        AppArtefact
```

Table 8. Description of AppScreenshot artefact class

For the last and most essential step of policy encoding we need to consider the policy conditions that go into the equivalent class axiom expression.

As an example, the CAST platform policy for app screenshots is as follows:

- File type should be either JPEG or PNG
- File size should 1024 KB or smaller
- Image should be between 300 pixels and 600 pixels high
- Image should be between 400 pixels and 800 pixels wide

In this specific instance, positive formulation is the only way to express the policy. This is because it is practically impossible to describe all the different conditions that would cause an app screenshot to be an invalid artefact. The opposite, however, is straightforward.

The equivalent class axiom expressing the policy conditions should therefore be attached to the class description for `ValidAppScreenshot`, converting it from a primitive OWL class into a defined class.

Table 10 provides the complete definition of `ValidAppScreenshot`. As shown, the policy conditions listed above are mapped onto the equivalent class expression in a direct way.

```
Class: ValidAppScreenshot

    EquivalentTo:
        (hasContentType some ({_image/jpeg, _image/png}))
        and (hasSizeInKB some xsd:integer[<= 1024])
        and (hasHeightInPixels some xsd:integer[>=300, <=600])
        and (hasWidthInPixels some xsd:integer[>=400, <=800])

    SubClassOf:
        AppScreenshot,
        hasContentType only ({_image/jpeg, _image/png}),
        ValidAppArtefact
```

Table 9. Description of positive-form policy ValidAppScreenshot (defined class)

The `EquivalentTo` expression is constructed as a conjunction of the different policy conditions. The expression comprises one object property (`hasContentType`) and three data properties (`hasSizeInKB`, `hasHeightInPixels`, `hasWidthInPixels`). The object property ranges over `_image/jpeg` or `_image/png`, which are ontology individuals serving as classification constants, as described in section 5.2.4. The data properties range over numeric integer values restricted by constraining facets (OWL 2 minInclusive and maxInclusive facets).

Last, notice the presence of the closure axiom for the `hasContentType` object property which is part of the `SubclassOf` expression of `ValidAppScreenshot`:

```
hasContentType only ({_image/jpeg, _image/png})        (1)
```

The purpose of the above axiom is to complement the `hasContentType` axiom that is part of the `EquivalentTo` expression and allow reasoners to draw the right inferences:

```
hasContentType some ({_image/jpeg, _image/png})        (2)
```

Axiom (2) describes the class of individuals that have *some* (i.e. at least one) connection along the `hasContentType` property to individual `_image/jpeg` or to individual `_image/png`. On the other hand, axiom (1) describes the class of individuals whose connections to any other individual along the `hasContentType` property are *only* with `_image/jpeg` or with `_image/png`. In other words, the individuals of the class described by closure axiom (1) do not have any connections along the `hasContentType` property to any individual whatsoever, except for individual `_image/jpeg` or individual `_image/png`.

The combination of the logical conditions explicated by the two axioms allows an OWL-DL reasoner like Pellet [127] or Hermit [128] to infer that a `ValidAppScreenshot` needs to be connected to either `_image/jpeg` or `_image/png`, and nothing else. In other words, the content type that any valid app screenshot individual can have is only JPEG or PNG, and nothing else.

Class `ValidAppScreenshot` is also defined as a subclass of `ValidAppArtefact`, allowing further chains of inference in modular definition of policies as we will see later in this section.

5.3.2.2 Negative formulation for artefact validation policies

Table 9 provides the definition of OWL class `InvalidAppScreenshot`. As shown, the description of the class includes nothing more than stating it is a subclass of `AppScreenshot` and `InvalidAppArtefact`. This means the `InvalidAppScreenshot` class description only states necessary but not sufficient conditions and is therefore a primitive class.

```
Class: InvalidAppScreenshot

    SubClassOf:
        AppScreenshot,
        InvalidAppArtefact
```

Table 10. Description of negative-form policy InvalidAppScreenshot (primitive class)

Because of the disjoint union axiom in the `AppScreenshot` class description, a DL reasoner is lead to deduce than any instance of an app screenshot artefact is either a `ValidAppScreenshot` or an `InvalidAppScreenshot`.

As a result, if an (asserted) instance of `AppScreenshot` satisfies the definition of `ValidAppScreenshot`, a DL reasoner will classify it as an (inferred) instance of `ValidAppScreenshot`. Conversely, if an (asserted) instance of `AppScreenshot` does not satisfy the definition of `ValidAppScreenshot`, it will be classified as an (inferred) instance of `InvalidAppScreenshot`.

The ability to draw these automated inferences provides the foundation for the policy evaluation infrastructure which is discussed in detail in section 7.

To offer one more example that illustrates how negative policy formulation works let us consider another CAST platform resource: `Description`.

`Description` is a data resource which is not represented as a file in CAST platform's repository but persisted and retrieved from a relational data store. It is a brief textual description available for every software unit on the CAST platform and, similarly to the above example of `AppScreenshot`, `Description` is also used in the cloud platform's app store/directory for end-users.

Figure 11 depicts the subsumption hierarchy.

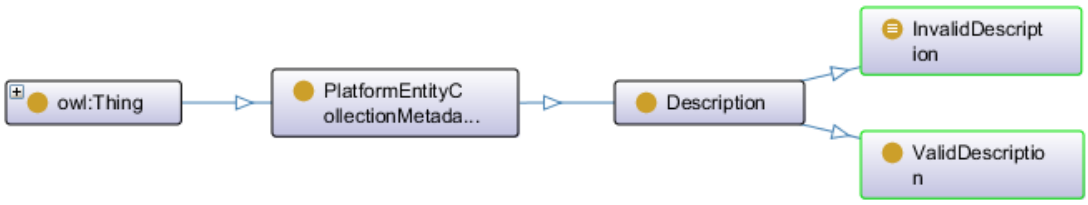

Figure 11. Class hierarchy in policy encoding pattern for the validation of Description resources

`Description` is subclass of `PlatformEntityCollectionMetadata`, which is a root class in the governance ontology (subclass of `owl:Thing`). A `Description` can either be a `ValidDescription` or an `InvalidDescription`.

The policy for valid descriptions of software units deployed on the CAST platform is very simple; the description must be non-empty text.

The defined OWL class for `InvalidDescription` which facilitates negative policy formulation is provided in Table 11.

```
Class: InvalidDescription

    EquivalentTo:
        Description
         and (hasSingleValue value "")

    SubClassOf:
        Description
```

Table 11. Description of negative-form policy InvalidDescription (defined class)

`InvalidDescription` is defined as the class of things which are known to be a `Description` and are also known to have a connection along the `hasSingleValue` data property to an empty string of characters.

Intuitively, any (asserted) instance of `Description` which does not satisfy the definition of `InvalidDescription` will be classified by a DL reasoner as an (inferred) instance of `ValidDescription`. Conversely, any (asserted) instance of `Description` that satisfies the definition of `InvalidDescription` will be classified as such.

5.3.2.3 Positive formulation for lifecycle management policies

The purpose of lifecycle management policies is to regulate the process by which software units developed by different ecosystem partners are integrated into the ecosystem's operational environment and continuously modified, ensuring integrity and consistency at all times. Process governance policy definition is tied to lifecycle stages and conditions under which a transition from one stage to another would be allowed. Similarly to what we discussed for artefact validation policies, the policy evaluation process is meant to reveal if a resource can move along the lifecycle via a transition to a new lifecycle state, or not.

A positive formulation of a process governance policy describes the conditions under which a transition from one state to another would be valid. Conversely, a policy expressed in negative form describes the conditions under which such a lifecycle stage transition would be invalid.

To illustrate the method of process governance policy encoding let us consider another example from the CAST policy dataset. On the CAST platform, software units can transition sequentially through the stages of development, testing, review, beta, production, deprecation, and end-of-life. Allowed transitions include promotion from one stage to the next but also demotion to the previous stage. Lifecycle stages are common between CAST platform software units (solutions, apps and services), but transition conditions differ depending on type of software unit.

Figure 12. CAST model of seven lifecycle stages of software units

The development state is the first state that a managed resource is entering once deployed to the platform. The testing state represents a phase during which a solution, app or service goes through platform integration testing in a special, non-production environment. The review state represents the phase of various automated and manual quality controls by platform QA experts to ensure certain levels of quality in an application. Beta represents a phase of acceptance testing by a wider group of users. The production state represents the phase where a service, app or solution is operational. Deprecation refers to the state of being possible to use but not recommended for usage, and end-of-life represents the state of being decommissioned from the platform.

Let us use the example of an app which is currently found in the Review stage (`AppInReview`) and a lifecycle management policy that dictates the conditions under which there can be a promotion of this app to the Beta stage.

The representation of process governance policies in positive or negative form is carried out in a pattern analogous to that employed for artefact validation policies. Policy encoding for `AppInReview` will proceed as follows:

1. Step 1: Create a *primitive* OWL class `AppPromotableToBeta`, which is asserted as a subclass of `AppInReview`. Should the governance policy be encoded in positive form, this class will be later converted to a *defined* class in step 4.
2. Step 2: Create a primitive OWL class `AppNonPromotableToBeta`, which is also asserted as a subclass of `AppInReview`. Should the governance policy be encoded in negative instead of positive form, it will be this class that will later be converted to a *defined* class.
3. Step 3: Create/modify the description of superclass `AppInReview`, asserting it is a subclass of `App` and a disjoint union of the newly created `AppPromotableToBeta` and `AppNonPromotableToBeta`.
4. Step 4: Depending on whether a positive or negative formulation is suitable, we construct an equivalent class axiom expressing the policy conditions and attach it to `AppPromotableToBeta` or to `AppNonPromotableToBeta`, respectively, converting one of the two into a defined OWL class.

Figure 13 illustrates the class subsumption relationships to be modelled, in order to govern the promotion of an app from one lifecycle stage to a subsequent one. In the example case we use here the goal is to control the lifecycle change of an app going from Review stage to Beta.

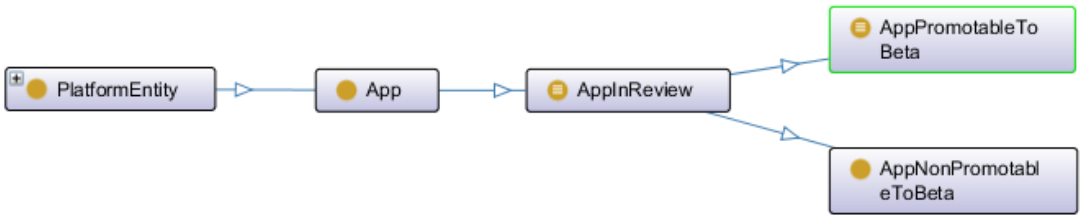

Figure 13. Policy encoding pattern to govern the promotion of an app from Review to Beta

`AppInReview` is an `App`, which is itself a `PlatformEntity`. An `AppInReview` can either be an `AppPromotableToBeta` or an `AppNonPromotableToBeta`. The OWL class description for `AppInReview` (as per step number 3 above) is provided in Table 12.

```
Class: AppInReview

    EquivalentTo:
        App
         and (hasLifecycleStateClassification value _Review)

    SubClassOf:
        App

    DisjointUnionOf:
        AppPromotableToBeta, AppNonPromotableToBeta
```

Table 12. Description of AppInReview stage

The CAST platform policy controlling the promotion of an app from Review to Beta stage states the following preconditions:

- The app has a non-empty text description
- The app's collection of artefacts includes a positive review report
- The app's collection of artefacts includes every artefact required for a transition to Review (i.e. is the app's current stage)
- The app's collection of artefacts are all valid with respect to the policies applicable
- The app has no dependencies to services, or in case it does, then all services this app depends on are of correct types and are in the stage of either beta or production

This policy is one more case where positive formulation is the only feasible way to express the conditions under which a lifecycle promotion would be valid. It is practically impossible to describe all the different conditions that would cause the promotion to be an invalid transition, so as to provide a formulation in negative form.

The equivalent class axiom expressing the policy conditions as per step number 4 should therefore be attached to the class description for `AppPromotableToBeta`, converting it from a primitive OWL class into a defined class.

Table 13 provides the complete definition of `AppPromotableToBeta`.

```
Class: AppPromotableToBeta

    EquivalentTo:
        AppInReview
        and (hasDescriptionMetadata exactly 1 ValidDescription)
        and (hasCollection exactly 1
               (CollectionOfValidAppArtefacts
                and AppArtefactsForTransitionToBeta))
        and (AppWithoutDependencies or
               ((hasDependency some
                  (PlatformEntityInBeta or
                   PlatformEntityInProduction))
                and AppWithDependenciesOfCorrectTypes))

    SubClassOf:
        AppInReview,
        hasDescriptionMetadata only ValidDescription,
        hasDependency only
            (PlatformEntityInBeta or
             PlatformEntityInProduction),
        hasCollection only
            (AppArtefactsForTransitionToBeta and
             CollectionOfValidAppArtefacts)
```

Table 13. Description of positive-form policy for AppPromotableToBeta transition (defined class)

In plain English, the `EquivalentTo` expression states that for an app to be promotable to Beta stage it must currently be in Review stage (`AppInReview`); must have a unique description which is valid with respect to the applicable policy (`ValidDescription`, as per Figure 11); must have a unique collection of artefacts which are all valid with respect to their applicable policies (`CollectionOfValidAppArtefacts`) and which includes all artefacts necessary for transition to Beta (`AppArtefactsForTransitionToBeta`); must not have any dependencies to other software units (`AppWithoutDependencies`), or in case it does, those units must be of the right type (`AppWithDependenciesOfCorrectTypes`) and those units must be in a lifecycle stage which is at least as advanced as the app's current stage (`PlatformEntityInBeta` or `PlatformEntityInProduction`).

One cannot fail but notice that the class description of `AppPromotableToBeta` is extensively modular and compositional, which is a strong advantage of ontology-based modelling frameworks. The equivalent class axiom (EquivalentTo) reuses several complex class descriptions which are described separately in the governance ontology:

```
ValidDescription
CollectionOfValidAppArtefacts
```

```
AppArtefactsForTransitionToBeta
AppWithoutDependencies
AppWithDependenciesOfCorrectTypes
PlatformEntityInBeta
PlatformEntityInProduction
```

In the interest of brevity we will not unfold each of these class descriptions here. The interested reader is referred to the ontology specification provided at www.ecosystem-governance.com.

We will only briefly comment on the construction of the OWL class descriptions for `CollectionOfValidAppArtefacts` and `AppArtefactsForTransitionToBeta`. The class `ValidDescription` has already been discussed in the previous section.

Table 14 provides the description of the `CollectionOfValidAppArtefacts` class.

```
Class: CollectionOfValidAppArtefacts

    EquivalentTo:
        (contains some ValidAppArtefact)
         and (contains only ValidAppArtefact)

    SubClassOf:
        AppCollection
```

Table 14. Description of CollectionOfValidAppArtefacts

The necessary and sufficient conditions of the class state that individuals belonging to this class must be connected along the `contains` property to at least one `ValidAppArtefact` (shown in Figure 10), and that all connections to anything along the `contains` property should be exclusively with individuals known to be a `ValidAppArtefact`. In other words, a collection should contain at least one app artefact and nothing else but valid app artefacts.

As discussed in the previous section, artefact validation policies that are expressed in positive form, like class `ValidAppScreenshot` for instance, are also specified as subclasses of `ValidAppArtefact`. The same holds for artefact validation policies in negative form, which are specified as subclasses of `InvalidAppArtefact`. This is to allow some very useful chains of inference with modular definition of policies.

For example, individuals which are asserted (i.e. statically declared) as instances of `AppScreenshot` and are also inferred (i.e. dynamically deduced) to be instances `ValidAppScreenshot`, will also be automatically inferred as instances of `ValidAppArtefact`, thus allowing a DL reasoner to evaluate the equivalent class expression in `CollectionOfValidAppArtefacts` above and produce the relevant deductions.

Table 15 below provides the description of `AppArtefactsForTransitionToBeta`.

```
Class: AppArtefactsForTransitionToBeta

    EquivalentTo:
        AppArtefactsForTransitionToReview
         and (contains some PositiveReviewReport)

    SubClassOf:
        AppCollection,
        contains exactly 1 PositiveReviewReport
```

Table 15. Description of AppArtefactsForTransitionToBeta

This is yet another compositional description that reuses OWL classes defined separately in the ontology. The necessary and sufficient conditions of the class state that the collections of app artefacts which should be inferred as belonging to this class must include all artefacts defined in the `AppArtefactsForTransitionToReview` class, and in addition, must contain a quality assurance review report with a positive evaluation.

In other words, the set of artefacts that an app should include in order to be ready for promotion to Beta stage is everything that was previously required for promotion to Review (the app's current stage), plus a review report that authorises deployment to the cloud platform's execution environment.

5.3.2.4 Negative formulation for lifecycle management policies

Table 16 provides the OWL class description for `AppNonPromotableToBeta`. As shown, the description of the class includes nothing more than stating it is a subclass of `AppInReview`. The `AppNonPromotableToBeta` class description only states conditions that are necessary (but not sufficient), and is therefore a primitive class. No further specification is necessary for `AppNonPromotableToBeta`. Because of the disjoint union axiom in the `AppInReview` class description, a DL reasoner is lead to deduce than any instance of an app screenshot artefact is either an `AppPromotableToBeta` or an `AppNonPromotableToBeta`.

```
Class: AppNonPromotableToBeta

    SubClassOf:
        AppInReview
```

Table 16. Description of negative-form policy for AppNonPromotableToBeta transition

Let us see one more example of lifecycle policy encoding in negative form. We will use another CAST platform lifecycle management policy: promoting an app from the stage of Deprecation (`AppInDeprecation`) to End-of-Life.

Figure 14 depicts part of the subsumption hierarchy involving the `AppInDeprecation` class.

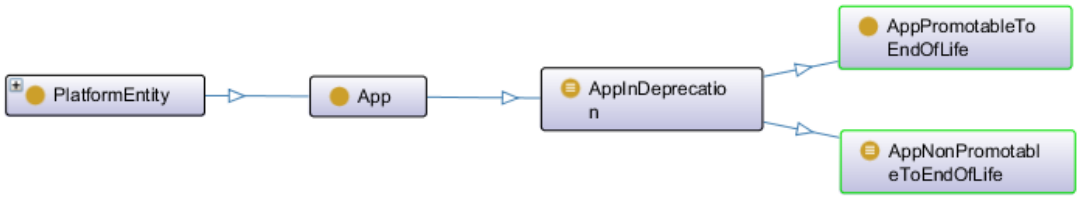

Figure 14. Policy encoding pattern to govern the promotion of an app to End-Of-Life.

`App` is subclass of `PlatformEntity`, which is a root class in the governance ontology. An `AppInDeprecation` is a subclass or `App` and can either be an `AppPromotableToEndOfLife` or an `AppNonPromotableToEndOfLife`.

The conditions under which an app can be promoted from Deprecation to End-of-Life (`AppPromotableToEndOfLife`) are quite basic:

- The app does not represent a dependency to any solution which is currently in Production stage

Intuitively, any app that has reached the Deprecation stage can be freely decommissioned from the platform's execution environment unless there are solutions which depend on this app and those solutions are not in the Deprecated stage too. The policy governing the lifecycle transition is therefore expressed in terms of those special/exceptional conditions which would cause the transition to be invalid (in negative form). Expressing the same policy in positive form would not be practical.

Table 17 provides the defined OWL class for `AppNonPromotableToEndOfLife`.

```
Class: AppNonPromotableToEndOfLife

    EquivalentTo:
        AppInDeprecation
         and AppWithDependentsInOperation

    SubClassOf:
        AppInDeprecation

    Rules:
        App(?a), Solution(?s), hasDependency(?s, ?a),
        hasLifecycleStateClassification(?s, _Production)
            -> AppWithDependentsInOperation(?a)
```

Table 17. Description of negative-form policy AppNonPromotableToEndOfLife

The equivalent class axiom above describes individuals that satisfy the conditions for membership in the `AppInDeprecation` class, and at the same time also satisfy the conditions for membership in the `AppWithDependentsInOperation` class.

The description of the `AppInDeprecation` class is straightforward, as shown in Table 18 below.

```
Class: AppInDeprecation

    EquivalentTo:
        App
         and (hasLifecycleStateClassification value _Deprecation)

    SubClassOf:
        App

    DisjointUnionOf:
        AppPromotableToEndOfLife, AppNonPromotableToEndOfLife
```

Table 18. Description of AppInDeprecation

However, the conditions for membership in the `AppWithDependentsInOperation` class are not equally straightforward to express in OWL. What we need to express involves chains of object properties and a non-tree shaped graph structure which is beyond the expressiveness capabilities of OWL 2. The conditions can be expressed with a first order predicate logic rule as follows:

```
App (a)
∧ Solution (s)
∧ hasDependency (s, a)
∧ hasLifecycleStateClassification (s, 'Production')
→ AppWithDependentsInOperation (a)
```

The first order logic rule presented above can be encoded in SWRL as shown in Table 19. The SWRL encoding can be appended to the OWL-based ontology encoding that an OWL-DL reasoner will later process to produce the desired inferences during policy evaluation.

```
(Class: AppWithDependentsInOperation)

    Rules:
        App(?a), Solution(?s), hasDependency(?s, ?a),
        hasLifecycleStateClassification(?s, _Production)
            -> AppWithDependentsInOperation(?a)
```

Table 19. Description of AppWithDependentsInOperation class membership conditions via a SWRL rule

As a result, Apps which are (asserted) instances of `AppInDeprecation` and have production-stage solutions that depend on them will be classified by an OWL-DL reasoner as instances of `AppNonPromotableToEndOfLife`. Conversely, (asserted) instances of `AppInDeprecation` which do not satisfy the definition of `AppNonPromotableToEndOfLife` will be classified as (inferred) instances of `AppPromotableToEndOfLife`.

5.4 Related work on semantic policy representation

5.4.1 Policy engineering

Fisler, Krishnamurthi and Dougherty [129] highlight a growing trend towards designing software applications in a way that certain rules are kept separate from the main program logic. Those rules are captured though policy-specification languages and consulted at run-time via an engine or reference monitor when user activity dictates to do so. In their view, this represents a new form of software modularisation that offers some "interesting twists to established software engineering problems". They observe that policies are much like other software components in their need for analysis, development, validation, and ongoing maintenance, and they propose to embrace policy engineering as a new method of thinking and problem solving in software systems design.

Lewis et al. [130] provide a definition for policy engineering as a "systematic approach to the development and maintenance of policies, which closely integrates the modelling of the managed system and its behaviour with capturing user goals and resolving them to system executable policies".

Ross-Talbot et al. point out that the area of policy languages is still maturing from both a research and industrial point of view [131]. A sign of the present immaturity is that policy, as a term, appears to be rather overloaded in relevant literature [132],[133]. Antoniou et al. [133] note that the term policy has been used in connection to several notions: (i) security (e.g. access control policies), (ii) trust management (policies for authentication on the basis of user properties in open environments such as the Web), (iii) action languages (reactive policy specification to execute actions such as authorisation), and (iv) business rules (formalising and automating business decisions). Bonatti and Olmedilla [132] also add quality of service (in the context of networks and distributed systems) to the list of notions which are considered relevant to the term.

Despite the lack of consensus on the semantics of the term 'policy', there has been a significant amount of work in this area, including research on how different aspects of policy engineering can be improved through the application of ontology-based knowledge representation and reasoning.

We can distinguish between two directions of research in this space. Research in the first direction has been mostly concerned with the development of novel ontology-based languages and tools for policy definition, management and enforcement, where the use of ontology-based knowledge representation and reasoning is central. Research works in the other direction have been mostly concerned with the enhancement of existing policy languages and tools with formal semantics and ontology-based methods of representation and processing.

5.4.2 Ontology-based policy representation and enforcement

The most prominent and representative works along the first line of research are KAoS [18] and Rei [126]. Tonti et al. [17] presents a comparison of the two systems and their approaches to policy representation, reasoning and enforcement, and compares them to Ponder [134], an earlier quite influential work in policy specification which did not make use of ontologies.

KAoS is a policy management framework relying on OWL and SWRL for the definition of policies [18],[135]. The framework comprises a set of core ontologies that can be extended per application domain, and a number of services supporting a variety of tasks related to creating, modifying and enforcing policies. The framework distinguishes between positive and negative authorisations (constraints that permit or forbid some action), and positive and negative obligations (constraints that require some action when an event occurs or that serve to waive such a requirement). Representing policies using OWL allows reasoning about the environment being controlled, policy relations, disclosure of policies, policy conflict detection, and policy harmonisation [136]. The reasoning infrastructure is provided by the Java Theorem Prover developed by the University of Stanford. The most notable domains in which KAoS has been applied are resource management in grid computing infrastructures and service-based workflows [137].

Rei is a related policy framework that integrates support for policy specification, analysis and reasoning in pervasive computing applications [17]. It is based on deontic concepts (although without any formal mapping to a particular deontic logic), and allows users to represent policies in terms of the concepts of rights, prohibitions, obligations and dispensations (i.e. deferred obligations) [126]. These modalities correspond to the notions of positive/negative authorisation and positive/negative obligation as found in KAoS [17]. The core concepts by which policies can be constructed are defined in an RDFS ontology. A policy engine makes decisions about authorisations and obligations by reasoning over policies using a Prolog-based reasoning engine. Before reasoning is executed, RDF triples are automatically translated to predicates of the form <*subject, predicate, object*> such that the Prolog engine can process them. One difference with respect to KAoS is that Rei has not been designed to enforce policies, but only to reason about them and respond to queries [17].

The Rein system that is presented in [138] represents follow-on work based on the original Rei design.

PolicyTab [139] is yet another approach for ontology-based policy representation with an emphasis on the aspect of trust negotiation and controlled access to resources on the Web. Policies are encoded using the PeerTrust policy language [140], with a distinction drawn between the notions of mandatory and default policies. F-Logic is used to formalise the constraints corresponding to those notions and to support reasoning.

5.4.3 Enhancing existing policy languages with formal semantics

Notable works along the second line of research mentioned earlier, i.e. the enhancement of existing policy languages and tools with ontological representation and reasoning, are those by Kolovski et al. [141] and Kolovski and Parsia [142], which focused on providing a mapping from WS-Policy to OWL. WS-Policy is a general purpose framework for describing capabilities of Web services and requirements of Web service consumers with a primary focus on non-functional properties. As with many other policy specification languages from the Web services domain (e.g. WSPL, XACML) the WS-Policy language lacks any formal semantics. By mapping the policy language constructs into a formal logic (Description Logics in this case) we can acquire a clear semantics for the language and obtain an understanding of the computational complexity involved in processing policies expressed in that language. Moreover, a mapping allows processing policies in that language using a general purpose DL reasoner, rather than a custom-built policy processor. Repeating the mapping process for additional policy languages can also enable reasoning about the exchangeability of policies represented in different languages and the interoperability of systems supporting them.

5.4.4 Discussion

Considering all this related work, it becomes apparent that the notions of policy which are adopted in the current literature are quite different from the notions of policy for process and resource governance that were presented earlier in this dissertation. We defined policy enforcement in the context of process governance as aiming to ensure that all resources relating to cloud services proceed through a prescribed set of lifecycle stages with well-defined transition criteria. In the context of resource governance, we defined policy enforcement as aiming to ensure that all resources associated with cloud services in the ecosystem satisfy certain conditions. In short, we appeal to a notion of governance policies that are meant to either constrain the evolution of cloud services throughout their lifecycle (lifecycle management policies), or constrain the structure and content of their associated resources (artefact validation policies).

This outlook is distinct from the views of policy that are commonly found in the literature, i.e. where policies are more narrowly associated with security, privacy, trust

management, quality of service, or business rules. However, it is consistent with broader definitions of the term, such as the one given by Tonti et al. [17] who describe policies as "*means to dynamically regulate the behaviour of system components without changing code and without requiring the consent or cooperation of the components being governed. By changing policies, a system can be continuously adjusted to accommodate variations in externally imposed constraints and environmental conditions*".

Most importantly, the benefits that ontology-based knowledge representation and reasoning approaches have introduced in the works presented above are still applicable, regardless of the fact that the domain of application is new. Tonti et al. [17] summarise the advantages afforded by ontology-based approaches to the representation and processing of policies to reduced human error, simplified policy analysis, reduced policy conflicts, and increased interoperability. Uszok et al. [18] add the advantages of reusability, extensibility, verifiability, safety, and reasonability.

5.5 Summary

In this chapter we present the first part of our implementation of the conceptual framework put forward in chapter 4. We discuss how to use a shared governance ontology to create policy definitions for process and resource governance in a cloud service ecosystem.

The previous chapter introduced the thesis that the architecture of software systems which are aimed at serving the advanced requirements of governance in cloud service ecosystems can benefit from Linked Data principles and Semantic Web standards to achieve their design objectives. The insight behind this thesis is that semantic technologies have already proven their value in analogous problem domains such as enterprise interoperability. Lessons learnt from that domain can be readily transferred to this new space in the form of architectural principles and concrete implementation technologies. On this basis the previous chapter introduced a conceptual framework architecture for governance support systems that realises this approach.

The proposed framework architecture for policy-driven governance in cloud service ecosystems (PROBE) comprises four core components: a shared governance ontology; a repository of ontology-based policy definitions; mechanisms to generate ontology-based resource descriptions; a governance policy evaluation engine.

This chapter opens with reiterating the view of the governance process from the viewpoint of the governance policy provider.

We then present the governance ontology that we created based on the governance support system requirements we had previously analysed in the scope of research

project CAST. The constructs described in our ontology mirror the structure and characteristics of the cloud services ecosystem researched in the scope of that project.

We introduced the different modelling constructs that the ontology makes available and presented a step-by-step method for creating governance policies. We put forward the view that any form of policy facilitating cloud service ecosystem governance can be abstracted to the level of either resource or process governance. In ecosystems facilitated by cloud application platforms, as in the case of CAST, process governance is effectively mapped onto platform lifecycle management, whereas resource governance is mapped onto platform artefact validation.

For some policies, it is more intuitive to express policy conditions in terms of what should necessarily hold in the domain (positive form), rather than what should not be the case (negative form). In other cases, it can be the opposite. And there are always cases where it is practically impossible to express a policy in one of the two forms. Acknowledging this, we put forward an ontology-based policy encoding pattern whereby process and resource governance policies can be formulated in either positive or negative form.

With the help of selected examples drawn from the CAST project policy dataset we presented detailed guidelines on encoding process and resource governance policies in both positive and negative form. We discussed the policy representation method in detail and highlighted how modular and compositional policy definitions can be, by virtue of our ontology-based approach. Lastly, we provided an overview of related work on semantic policy representation and some of the literature highlighting the advantages of policy engineering with formal ontology-based semantics.

The key takeaways from this chapter can be summarised as follows:

1. Our analysis to date supports that any form of policy facilitating cloud service ecosystem governance can be abstracted to the level of either resource or process governance. Process governance is concerned with ensuring a disciplined approach to introducing, modifying or removing software units from the ecosystem. Resource governance is concerned with ensuring that all artefacts and metadata linked to software units conform to the relevant technical, business or legal constraints.
2. Constraints in governance policies can be expressed in terms of either what should hold in a situation (positive formulation) or what should not be the case (negative formulation). Situations where both types of formulation are equally applicable are uncommon – it is usually clear that one type of formulation is more straightforward and preferable over the other. This applies to both resource and process governance policies.
3. We describe the conceptualisation and representation of an ontology that serves as a shared ecosystem vocabulary to describe governed software resources, and

at the same time also provides the vocabulary to define the necessary types of governance policies. The method is sufficiently expressive to allow describing diverse forms of cloud service resources and policies, covering governance objectives ranging from strategy to operations, and descriptions ranging from pricing models to lifecycle transitions. It is also sufficiently expressive to represent both types of governance policies (process and resource governance), as well as both positive and negative formulation of constraints.

4. Because of its foundation on the Web Ontology Language (OWL) standard and related Semantic Web technologies, the proposed method of defining governance policies is readily equipped to support heterogeneity, distribution and continuous evolution. It is natively suited to support the type of networked collaboration found in cloud service ecosystem governance. It allows decoupling governance functions by offering a way to describe the policy conditions separately from the governance subjects and the policy evaluation logic. It ensures interoperability by offering a platform-agnostic way for ecosystem partners to exchange/share policies and data over the internet. It increases abstraction, by allowing ecosystem partners to bridge their terminology spaces to a common ecosystem-level vocabulary.
5. By virtue of OWL's declarative encoding style and its formal logic underpinnings, our proposed method also facilitates advanced automation in policy engineering tasks, such as traceability of logical dependencies between policies, detection of contradictions with other policies and debugging of complex policy logic (e.g. through satisfiability tests). More generally, the unambiguous interpretation and automated reasoning capabilities afforded by OWL's formal semantics fulfills the need of increased operational efficiency - through improved maintainability, reusability, traceability and overall agility.

Chapter 6

Describing governed resources

6 Describing governed resources

6.1 Introduction

For process and resource governance to be feasible through a generic and universal ontology-based method, it is not only the policies, but also the heterogeneous ecosystem resources which are subject to governance that must be described in an abstract and homogeneous manner. These descriptions need to be extracted from the multiple forms in which cloud platform resources are natively represented and persisted, to create Linked Data, using the governance ontology as common reference vocabulary for the domain.

The term Linked Data refers to a set of best practices for publishing and connecting structured data using key Web technologies: URI, HTTP, and RDF. Using Linked Data principles and relevant web standards allows us to represent governance subjects in a way that is independent from how policies are encoded or how policy evaluation engines operate. In fact, the usage scenarios for the Linked Data which can be produced through the process we will describe can include many more applications beyond governance policy enforcement.

In this chapter we present one possible way of realising the third component of the conceptual framework we proposed in chapter 4, i.e. the mechanisms to generate and share ontology-based descriptions of governed ecosystem resources. A wide range of academic and commercial efforts in the field of Linked Data have recently provided several tools which can be used for this purpose. We will show examples of how governed resources can be described, once again based on the governance policy dataset from project CAST. Lastly, we will briefly discuss related work on Linked Data enablement tools and application architectures.

6.1.1 Governance from the resource provider's perspective

As discussed in chapter 3, the governance data provider role is responsible for creating, maintaining, and providing information about resources which are available in a software ecosystem and are subject to governance. Typically, the provider of this information will also be the actor that owns or manages the relevant resource.

To offer some examples, let us refer back to the ecosystem governance scenarios from chapter 3. The role of governance data provider in scenario #1 is assumed by the application developers (internal staff or external partners) who create apps and submit them to NineLives for quality review. In scenario #2 this role belongs to the compliance management team at NineMed. In scenario #3 the data provider role belongs to the ISVs who create and submit applications for deployment to the CloudDev platform, but also to CloudDev itself – once an application has been successfully deployed to the

platform. In scenarios #4 and #5 the role is again assumed by CloudDev since the governed resources (i.e. the applications developed by ISVs) have transitioned under the management and control of CloudDev.

Providing ecosystem partners with information regarding the governed resources is essential in order for the different policies relating to those resources to be evaluated. The information about a governed resource could be primary data residing in files or databases, or data that is extracted from primary sources specifically for the purpose of policy evaluation.

The concerns of individual providers of governed resources are analogous to those of the policy providers: how to effectively and efficiently manage governed resource descriptions internally, and how to easily communicate these to other ecosystem partners.

- Internal management objectives: To be able to freely make changes to existing governed resource data, its schema, formats or the systems through which the data is internally persisted and managed, while containing changes locally, i.e. without necessitating any corresponding changes to third-parties such as policy providers or policy evaluators.
- External communication objectives: To be able to share governed resource data in an effortless way, without needing to consider how the data will later be processed by policy evaluators, or how policies concerning the governed resources are represented.

6.1.2 Governed resources from the ecosystem's perspective

Observed from an ecosystem-wide perspective, descriptions of governed resources exhibit the following three characteristics.

Heterogeneity:

- They concern very different aspects of a software ecosystem resource, from the pricing model or localisation details of a software unit, to its lifecycle stage.
- They are represented in very dissimilar native formats (XML files, non-XML configuration files, relational databases, scripts, source code).

Physical distribution:

- They are stored in different locations. Software ecosystem partners are distributed, and so is the data they hold regarding the governed ecosystem resources.
- They need to be exchanged over the internet between ecosystem partners.

Fragmented ownership and control:

- Governed resources are owned by multiple different ecosystem partners who are free to evolve (modified, removed) independently of other governance process stakeholders.
- Governed resources are owned by ecosystem partners who are free to choose and use their own terms of reference in their local/native resource descriptions.

In an environment where the data relating to governed resources is heterogeneous, distributed and under multiple different ownership domains, it is imperative to standardise formats for data exchange. This is the only way to achieve loose coupling between the resources being governed, the governance policies and the policy evaluation mechanisms.

Without standardisation there would be no option but to implement policy evaluation mechanisms such that they work with:

(i) the specific data formats/sources that are imposed by each governance data provider (for instance, one resource provider may offer descriptions of services based on the WSDL 1.1[48] standard while another provider uses WSDL 2.0[49]);

(ii) the proprietary APIs for accessing governance subject data that are imposed at each data provider's site (e.g. a vendor-specific API for fetching a WSDL document from a particular data provider's server).

This approach is not feasible as it makes the owner of a policy evaluation engine entirely dependent on each and every one of the governance data providers in the ecosystem. It means that the implementers of policy evaluation engines need to understand the terms of reference that each governance data provider is using when describing their resources. It also means that should a governance data provider decide to make a change to their proprietary data formats or to their APIs, the operation of every policy evaluation engine that is coupled to these technical specifications is affected.

Such a model of policy-based governance would not scale for ecosystems.

With standardisation in place, the setting is different. The policy evaluation mechanisms of different stakeholders can be built to consume a single, common data representation of governed resources. Commitment is made only to a single specification, rather than to several specifications. There is a single terminology that developers and operators of policy evaluation engines need to understand. Any internal changes to how resource description providers manage their data will remain localised to individual providers, having no impact on the operation of the policy evaluation engines. Only changes to the common data representation format would necessitate extensive reengineering of the policy evaluation engines.

[48] https://www.w3.org/TR/wsdl
[49] https://www.w3.org/TR/wsdl20/

6.2 Descriptions of governed resources as Linked Data

The foundation to achieve the required standardisation and provide platform-agnostic descriptions of governed resources can be readily offered by Linked Data principles, Semantic Web standards, and related tools for data extraction and publishing.

6.2.1 Linked Data principles

As introduced in section 4.3, Linked Data refers to a set of best practices for online sharing of structured data in the form of interlinked datasets.

Foundational web technologies such as HTTP and URIs are employed by Linked Data and coupled with new technology standards such as RDF and SPARQL to enable the weaving of a global distributed database.

For completeness, we repeat here the four Linked Data principles defined by Berners-Lee to guide the publishing of structured interlinked data [114]:

1. Use URIs as names for things
2. Use HTTP URIs so that people can look up those names
3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL)
4. Include links to other URIs, so that they can discover more things

6.2.2 Core Semantic Web standards

The core enabling technologies of the Semantic Web, beyond the foundation of HTTP and URI, are RDF, OWL and SPARQL. These standards were briefly introduced in chapter 4, and examples of OWL usage were provided in chapter 5. In the sections below we will briefly introduce RDF and SPARQL before moving on to present an example of creating governed resource descriptions based on Linked Data principles.

6.2.2.1 RDF

With RDF (Resource Description Framework), resources are described by creating statements in the form of subject–predicate–object expressions. These expressions are known as RDF triples.

The subject in an RDF triple denotes the resource being described. The predicate may denote an attribute of that resource (a data property), or a relationship between the resource and some other resource (an object property). The object will correspondingly denote the data value of the resource's attribute, or denote the resource to which the described resource is associated [143].

Subjects and predicates in an RDF triple are uniquely identified and represented using URIs. When objects in an RDF triple denote relationships to other resources they are also represented by a URI. When objects denote an attribute of the described resource (a data property), they are represented as a literal value.

The triples combined form an RDF graph, as shown in Figure 15 below.

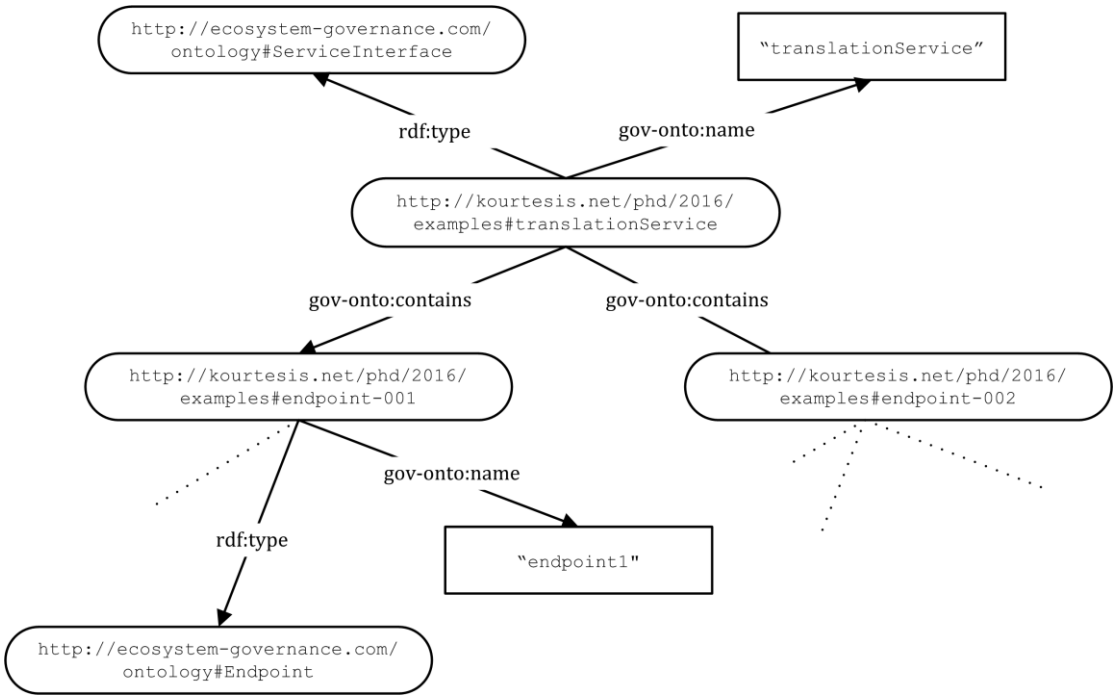

Figure 15. Example RDF graph.

6.2.2.2 SPARQL

SPARQL (SPARQL Protocol And RDF Query Language) defines a query language for data that is represented in the directed, labelled graph data format provided by the RDF standard. It has become the predominant query language for RDF graphs. The language can be used to express SQL-like queries across diverse data sources, whether the data is stored natively as RDF or is viewed as RDF via some kind of middleware [113]. By virtue of OWL's layering on top of RDF, SPARQL can also be used to query instance data (individuals) defined in an OWL ontology.

In summary, SPARQL allows developers of Linked Data applications to perform the following [23]:

- Pull values from structured and semi-structured data
- Explore data by querying unknown relationships
- Perform complex joins of disparate databases in a single, simple query
- Transform RDF data from one vocabulary to another

6.3 Method for creating and sharing descriptions of governed resources

6.3.1 Examples of governance data description

As mentioned earlier, the abstract data description framework provided by RDF has many applications, allowing structured and semi-structured data in different ownership domains to be easily exposed and shared across organisational boundaries and heterogeneous applications. Because of these characteristics, RDF also provides a viable foundation to describe and to share descriptions of governed resources in the context of a cloud services ecosystem.

Let us look at an example of describing a governed resource. Table 20 provides a snippet from an XML document describing a service available on the CAST platform. It is excerpted from a web service interface description artefact encoded using the WSDL 2.0 standard (Web Services Description Language). The service described can be used by apps on the CAST platform to translate text between different pairs of languages.

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns="http://www.w3.org/ns/wsdl"
    targetNamespace=
"http://cast-project.eu/governance/examples/wsdl"

[...]

  <service name="translationService"
       interface="tns:translationServiceInterface">

     <endpoint name="endpoint1"
               binding="tns:translationSOAPBinding"
               address ="http://144.76.8.88"/>

     <endpoint name="endpoint2"
               binding="tns:translationSOAPBinding"
               address ="http://143.167.8.2"/>
  </service>

</description>
```

Table 20. Excerpt from translation service interface description (CAST platform WSDL artefact)

In the WSDL excerpt above we can see that the interface of `translationService` includes two endpoints with different HTTP addresses: `endpoint1` is accessible at address `http://144.76.8.88` and `endpoint2` is accessible at `http://143.167.8.2`.

This is one piece of information which is relevant from a governance perspective.

One of the artefact validation policies from the CAST policy dataset states that the interface specification (WSDL 2.0 document) of every external web service used by apps on the CAST platform should specify two unique endpoint URLs. These URLs should point to different servers through which the service can be provided (primary and backup endpoints). The rationale is to provide a failover alternative in case the primary server that hosts the service becomes unavailable.

As per our framework and proposed method, the `translationService` interface can be validated against this policy as long as the information provided in the WSDL document presented in Table 20 above is first extracted from the document, and made available in a suitable RDF-based representation. This is a task to be performed by the relevant actor that assumes the role of governance data provider.

This representation can be derived from the WSDL document programmatically and will include the facts shown in Table 21, expressed as subject-predicate-object RDF triples. This is a process that is commonly referred to as "triplification" and there is an abundance of Linked Data tools and software frameworks that one can reuse for this purpose which are discussed later in this chapter.

```
S:    <http://kourtesis.net/phd/2016/examples#translationService>
P:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
O:    <http://ecosystem-governance.com/ontology#ServiceInterface>

S:    <http://kourtesis.net/phd/2016/examples#translationService>
P:    <http://ecosystem-governance.com/ontology#contains>
O:    <http://kourtesis.net/phd/2016/examples#endpoint-001>

S:    <http://kourtesis.net/phd/2016/examples#translationService>
P:    <http://ecosystem-governance.com/ontology#contains>
O:    <http://kourtesis.net/phd/2016/examples#endpoint-002>

S:    <http://kourtesis.net/phd/2016/examples#endpoint-001>
P:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
O:    <http://ecosystem-governance.com/ontology#Endpoint>

S:    <http://kourtesis.net/phd/2016/examples#endpoint-002>
P:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
O:    <http://ecosystem-governance.com/ontology#Endpoint>

S:    <http://kourtesis.net/phd/2016/examples#endpoint-001>
P:    <http://ecosystem-governance.com/ontology#contains>
O:    "http://144.76.8.88"^^<http://www.w3.org/2001/XMLSchema#anyURI>

S:    <http://kourtesis.net/phd/2016/examples#endpoint-002>
P:    <http://ecosystem-governance.com/ontology#contains>
O:    "http://143.167.8.2"^^<http://www.w3.org/2001/XMLSchema#anyURI>
```

Table 21. Raw RDF triples extracted from translation service WSDL artefact

The RDF triples presented above could be serialised in static RDF/XML files and placed on a web server which is controlled by the data provider. Ideally though, they should be persisted in an RDF triple store which exposes a SPARQL query interface. In the latter case, any (ecosystem-authorised) governance policy evaluation engine could query the RDF triple store and retrieve the relevant information on demand.

Table 21 presents a SPARQL query that can be used to obtain all information which is part of the RDF graph at the governance data provider's end, and relates to the service of interest (`translationService`).

```
SELECT DISTINCT ?predicate ?object
WHERE
{
   <http://kourtesis.net/phd/2016/examples#translationService>
   ?predicate
   ?object
}
```

Table 22. SPARQL query to retrieve RDF description of translation service interface

The RDF triple store to receive a SPARQL query similar to that shown above would return a document as shown in Table 23. In this example, the encoding of RDF triples is in Turtle syntax [144].

```
@prefix : <http://kourtesis.net/phd/2016/examples#> .
@prefix gov: <http://ecosystem-governance.com/ontology#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@base <http://kourtesis.net/phd/2016/examples> .

:translationService  rdf:type  gov:ServiceInterface ;
:translationService  gov:contains  :endpoint-001 ;
:translationService  gov:contains  :endpoint-002 ;

:endpoint-001  rdf:type  gov:Endpoint ;
:endpoint-001  gov:hasAddress  "http://144.76.8.88"^^xsd:anyURI ;

:endpoint-002  rdf:type  gov:Endpoint ;
:endpoint-002  gov:hasAddress  "http://143.167.8.2"^^xsd:anyURI .
```

Table 23. RDF description of translation service interface encoded in Turtle syntax

For the rest of the RDF description examples in this dissertation we will be using Turtle (Terse RDF Triple Language) syntax. Turtle is a textual syntax for RDF that allows RDF graphs to be completely written in a compact and natural text form, with abbreviations for common usage patterns and datatypes [144].

Turtle's `@prefix` directive allows declaring a short prefix name in place of a long URI prefix. Writing `@prefix gov: <http://ecosystem-governance.com/ontology#>` allows us to subsequently write `gov:ServiceInterface` and having this expression

interpreted as `http://ecosystem-governance.com/ontology#ServiceInterface`. Similarly, the `@base` directive allows creating a default prefix.

6.3.2 Linked Data provision & sharing architecture

The RDF description presented in Table 23 is equivalent to that in Table 21, and conveys the same information as presented in the WSDL document of Table 20. This is a case where structured data in the form of a single WSDL/XML description can be transformed into a structured RDF dataset. In other cases however, the RDF description of a governed resource may need to be constructed by aggregating, assembling and transforming data and generating RDF statements from a number of different sources.

The sources of primary data regarding governed resources will typically be:

- Databases (e.g. relational databases, document/NoSQL databases)
- APIs (e.g. interfaces of registry & repository systems)
- Files (e.g. XML specification/property files, binary file headers)

For example, the RDF description of a CAST platform app would include a fairly large set of statements regarding each of the artefacts associated with the app. This could include the app deployment descriptor, properties, localisation, license, pricing, provider details, description, iconography, review report and several more, but also metadata such as the lifecycle stage of the app, its dependencies on ecosystem services, etc.

There is a great degree of heterogeneity in the native data sources, but RDF and SPARQL allow us to abstract over the differences and provide a common description layer for all resources associated with the CAST platform app in question, or any governed resource in a software ecosystem.

In their book on design patterns for Linked Data applications, Heath and Bizer [19] note that despite the large number of information systems that can be connected into the "Web of Data", the mechanisms for doing so fall into three Linked Data publishing patterns.

These publishing patterns are:

1. Generating Linked Data from queryable structured data. A relevant example from the CAST policy dataset would be to query the relational DB of the CAST platform governance support system (CAST Registry & Repository) to retrieve lifecycle state information on an app (e.g. to enforce an end-of-life policy) and generate the respective RDF statements.
2. Generating Linked Data from static structured data. We have already seen the example of generating an RDF graph from the WSDL document of Table 20.

Other relevant examples from CAST platform governed resources could be the XML pricing specification file or the app deployment descriptor.

3. Generating Linked Data from unstructured data. An example could be to generate a structured RDF graph from a text document. This could be useful in the context of analysing a document and carrying out a process of named entity extraction. However, in our experience, this does not appear to be common in the context of a governance support system where enterprise data is usually structured.

The variety of possible workflows as identified by Heath and Bizer is visualised in Figure 16 below.

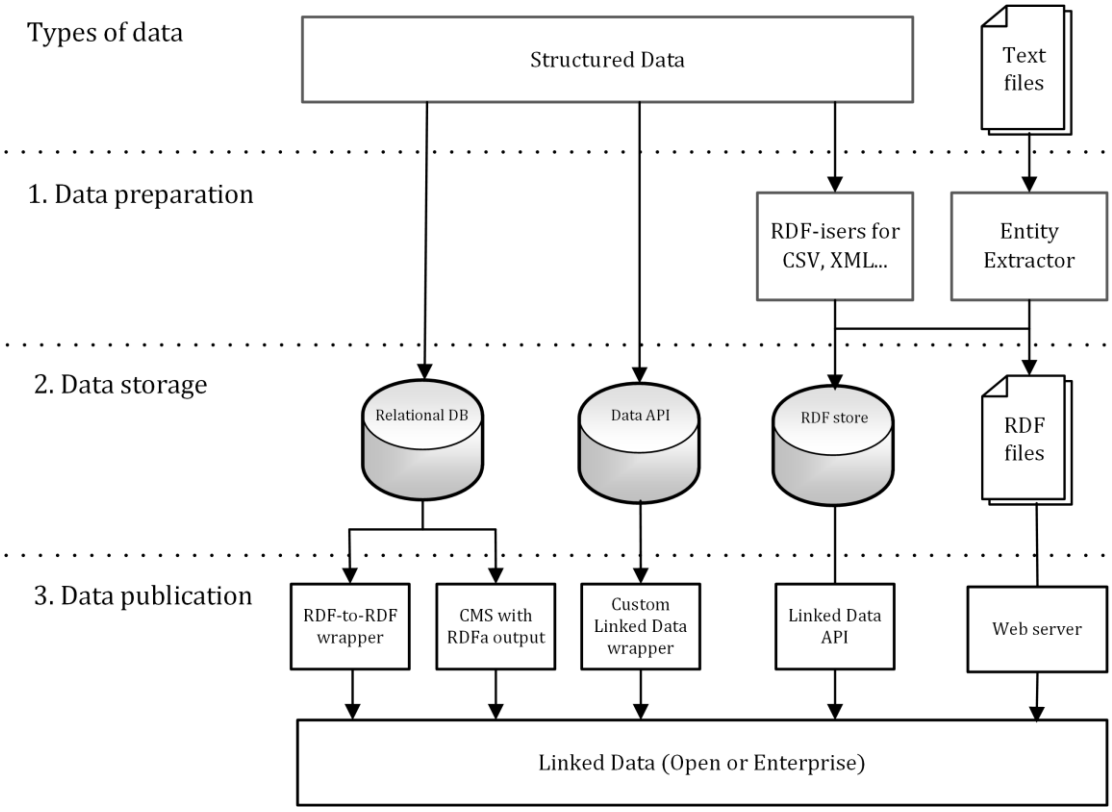

Figure 16. Linked Data publishing options and workflows. Adapted from [19].

In the context of governance, the most common scenarios would concern publishing RDF data from structured sources. As mentioned these could be databases, APIs or static files. In all cases, the architectural solution would involve some form of wrapper components that perform the transformation from the native data representation formats to RDF statements. These transformations need to be driven by mapping specifications, either unidirectional or bidirectional, which can guide a machine to automate the process.

Fortunately, such an architecture solution would not need to be developed from the ground up. There is already an abundance of software frameworks, platforms,

programming libraries and tools, most of which are open-source, which can provide the basis for an implementation.

The majority of creators of tools in this space have so far focused on facilitating the generation of Linked Data from queryable structured data that reside in relational databases. This was motivated by very practical reasons, as the majority of data around the world currently resides in SQL databases. NoSQL (which stands for Not Only SQL) databases that have reached popularity more recently are also supported by several RDF enablement tools, such as those focusing on generating RDF graphs from Key-Value stores and other non-relational structures [145].

6.3.2.1 Translation approaches and requirements for governance support

Michel et al. [146] have carried out a thorough survey of RDB to RDF translation approaches and technologies. They propose classifying RDB to RDF systems with respect to the following three dimensions:

1. The way mappings from RDB to RDF are described (direct vs transformational mapping specifications)
2. The way mappings are implemented to generate RDF data (static vs on-demand production of RDF triples)
3. The way RDF data is being accessed by applications (query-based access vs URI-based access)

We will briefly walk through each of the above dimensions and discuss which approach is most suitable from an ecosystem governance perspective.

Mapping specification

RDB to RDF mapping specifications can be provided in a direct or transformational manner.

- The direct mapping approach converts relational data into RDF in a straightforward fashion, by applying generic translation rules (defined by Tim Berners-Lee since as early as 1998 [147]): table to class, column to property, row to resource, cell to literal value or to resource URI. A by-product of the process is an ad-hoc RDF vocabulary (ontology) that mirrors the relational schema.
- Conversely, the transformational/custom mapping approach is applied when the relational database needs to be translated using concepts and properties from existing ontologies. A typical use case is the alignment of a legacy database with an existing ontology that describes the same domain of interest. The relevant recommendation by W3C is R2RML – a language for expressing customised mappings from relational databases to RDF datasets [148].

Mapping implementation

Relational database tuples can be translated into RDF statements or ontological instances (individuals) through either data materialisation or on-demand mapping.

- Data materialisation represents the static transformation of the database into an RDF representation, similarly to an Extract-Transform Load (ETL) approach. Mapping rules are applied to contents of the database to create an equivalent RDF graph. The resulting triples can be loaded into an RDF triple store and accessed through the triple store's SPARQL query interface.
- The on-demand mapping approach allows run time evaluation of queries against the relational data. In other words, the RDF dataset is virtual and gets constructed dynamically. SPARQL queries (or any other form of query) to retrieve data from the RDF graph are translated into SQL at query evaluation time. The relational data is never really transformed, just translated.

Data retrieval

The RDF statements about a described resource can be retrieved through either a SPARQL query, or by dereferencing the URI of the said resource, and performing content negotiation with the Web server.

- SPARQL-based access: When the mapping implementation follows data materialisation as an approach, the SPARQL endpoint evaluates a query against the RDF triple store in which the materialised RDF triples have been loaded. In the case of the on-demand mapping approach, the SPARQL endpoint rewrites the SPARQL query into SQL queries and, vice-versa, translating SQL results into an equivalent SPARQL response.
- URI-based access: Every resource described through a set of RDF triples is assigned a URI as unique identifier. According to the principles of Linked Data that we discussed in section 6.2.1, these URIs are possible to dereference by performing an HTTP GET method, as if the URI was a URL. Through content negotiation the Web server can then be instructed to return an RDF dump or to present additional information about the resource in a different form (such as human-readable HTML).

Based on the above analysis, and in the context of our proposed framework for ecosystem governance support systems, it appears that a certain combination of mapping specification, mapping implementation and data retrieval modalities would be more suitable than other combinations.

In our context, the governance ontology is already provided as a common vocabulary for describing governed resources and defining governance policies. Therefore, a transformational approach to specifying custom mappings between RDF and RDB would be the most appropriate route, as opposed to a direct mapping approach which

would result in the production of a new ontology model from every different data source being mapped.

In terms of mapping implementation, an on-demand mapping approach whereby a virtual RDF graph is constructed dynamically is more suitable than a static materialisation approach. The main reason is managing updates to the RDF graph. In the context of a dynamic multi-party ecosystem, one would expect frequent updates to the (relational) data relating to governed resources, which means a materialised RDF graph could quickly become outdated. Data materialisation is not the optimal approach in such contexts [146].

Lastly, as far as data retrieval is concerned, a SPARQL-based approach to accessing the generated RDF dataset seems more practical, for reasons having to do with the implementation of the policy evaluation mechanism, to be discussed in the following chapter.

6.3.2.2 State of the art implementation options

Technologies that address the above requirements and could be readily employed to support implementations of our governance support system framework include D2RQ [149], Oracle 12c [150], Virtuoso [151], Optique [152] and Quest/Ontop [153],[154],[155]. All of the above frameworks support custom transformational mappings from relational schema to an ontology, and most of them do so by supporting the R2RML standard by W3C [148]. Moreover, they all support dynamic generation of RDF and ease of querying over virtual RDF view models through SPARQL.

It is worth noting that RDB to RDF systems are continuously evolving with significant performance breakthroughs in recent years. For instance, the recently released Quest engine which is bundled with the Ontop platform implements a new design approach and delivers performance 500x times faster than D2RQ, and an average of 10x faster than Virtuoso[50]. Oracle's investment into creating a commercial RDF graph store solution on top of its Oracle database range is also worth mentioning.

Beyond relational data, the above mentioned RDB to RDF systems can also support non-relational structured sources (NoSQL databases, static structured documents, or APIs). This is done by supporting an extensible wrapper architecture which allows source-specific wrappers to expose the data from these sources as if they were relational, as an intermediate step before the conversion to RDF is performed.

The availability of this range of tools is partly attributed to years of research in the field of Ontology-based Data Access (OBDA) [145],[156]. The fundamental idea in OBDA is to provide users with more convenient access to data residing in traditional databases,

[50] http://ontop.inf.unibz.it/components/ontopquest/

without exposing the complexity of the raw data sources. This is achieved through the use of an ontology.

Queries against the data are formulated using the vocabulary provided by the ontology, without the user ever having to know the actual structure of the data. The relationships between the ontological vocabulary and the data schema are described by OBDA mappings. OBDA user queries are first enriched using logical reasoning by compiling relevant parts of the ontology into the query, and then unfolded, i.e., translated into SQL queries using the OBDA mappings [152]. The approach is often referred to as "end-user oriented data access" or "user-oriented view" [156], and its fundamental affordance is that it allows domain experts to express information needs in their own terms.

6.3.2.3 Further applications of semantic descriptions for ecosystem resources

Notably, the above-described way in which Linked Data for governed resource descriptions can be represented is not determined by how policies are encoded or how policy evaluation engines operate. In fact, the usage scenarios for the Linked Data which is produced through this process can include much more than just governance policy enforcement.

A stream of research in cloud computing has recently been looking into applications of Linked Data - and semantic technologies overall - for improving systems management in cloud environments. Haase et al. [157] describe the challenges related to intelligent information management in enterprise clouds and discuss how semantic technologies have been leveraged to address those challenges in the commercial eCloudManager system developed by fluidOps.

In [158], Feridun and Tanner from IBM describe an approach and architecture for the transformation of diverse network and server management data into Linked Data, which allows data-center operators to easily browse, search and query data across multiple sources. Also, Joshi [159] describes some initial work towards a policy-based framework facilitating the automation of the lifecycle of virtualised services, using ontologies and Semantic Web technologies like OWL, RDF and SPARQL.

6.4 Summary

In this chapter we presented the second part of our implementation of the conceptual framework put forward in chapter 4. We discussed how to describe ecosystem resources which are subject to governance in a way that will later enable evaluation against the ontology-based policy definitions described in chapter 5.

The actor owning or managing a software resource which is made available to the ecosystem is also responsible for creating, maintaining, and providing access to information about that resource for the purposes of policy enforcement. The primary data from which this information is to be extracted could reside at a single location or at multiple different locations, collected from databases, specification files or APIs.

From an ecosystem point of view, the governed resource descriptions that need to be contributed by different stakeholders are highly heterogeneous, highly distributed, and span across organisational boundaries. Evidently, in such an environment it is imperative to standardise semantics and formats for data exchange. The foundation to achieve the required standardisation and provide platform-agnostic descriptions of governed resources can be readily offered by Linked Data principles, Semantic Web standards, and related tools for data transformation and publishing.

Through examples, we presented a method to create RDF descriptions of governed resources using the vocabulary provided by the governance ontology introduced in chapter 5. Often, these RDF descriptions of software resources need to be constructed by aggregating, assembling and transforming primary data from a number of different structured sources. A requirement emerges for an infrastructure that allows mapping the native data representations that are relevant to those resources to high-level concepts in an ontology-based vocabulary, and then generating the RDF descriptions.

Fortunately, such an infrastructure would not need to be developed from the ground up, as there is an abundance of software RDB to RDF frameworks which can provide the basis for an implementation. These frameworks employ different approaches for describing mappings, generating RDF triples and providing access to the RDF data. In the context of our proposed framework for developing ecosystem governance support systems, the ideal approach combines transformational mappings against our governance ontology, dynamic on-demand generation of RDF triples and SPARQL-based access.

The key takeaways from this chapter can be summarised as follows:

1. The ecosystem partner who owns or manages a software resource is also responsible for creating, maintaining, and providing access to information about that resource for the purposes of policy enforcement. This data can be heterogeneous, physically distributed and under multiple different ownership domains. It is therefore imperative to standardise formats for data exchange.
2. Employing a Linked Data approach for the description of governance subjects achieves a loose coupling between the resources being governed, the governance policies and the policy evaluation mechanisms. Applying a Linked Data approach means that governance data providers: (a) use URIs as names for governed resources, (b) use HTTP URIs so that entities acting as policy evaluators can look up those names, (c) provide information about the governed

resources in RDF when a URI is looked up with a SPARQL query, (d) include links to other URIs in the information returned to a SPARQL query, so that ecosystem partners can further discover more useful information.

3. The abstract data description framework provided by RDF has many applications, allowing structured and semi-structured data in different ownership domains to be easily exposed and shared across organisational boundaries and heterogeneous applications. Because of these characteristics, RDF also provides a viable foundation to describe and to share descriptions of governed resources in the context of a cloud services ecosystem. These descriptions can be derived from their native data sources programmatically, through a process that is commonly referred to as "triplification" and is supported by a wide range of software tools.
4. Data triplification approaches can be classified with respect to (1) how mappings from RDB to RDF are described (direct vs transformational mapping specifications), (2) how mappings are implemented to generate RDF data (static vs on-demand production of RDF triples), (3) how RDF data is being accessed by applications (query-based access vs URI-based access). In the context of ecosystem governance support systems the preferred approach combines transformation mappings between RDF and RDB, on-demand data translation where RDF graphs are constructed dynamically and SPARQL-based access the generated RDF dataset.
5. There is a great degree of heterogeneity in the native data sources, but RDF and SPARQL allow us to abstract over the differences and provide a common description layer for all resources associated with any governed resource in a software ecosystem. The usage scenarios for the Linked Data which is produced through this process can include much more than just governance policy enforcement.

Chapter 7

Evaluating governance policies

7 Evaluating governance policies

7.1 Introduction

In the previous two chapters we went through the method to create policy definitions based on our governance ontology, and descriptions of governed resources based on Linked Data principles.

In this chapter we present the last part of our implementation of the PROBE framework as put forward in chapter 4. We briefly revisit the role of the policy evaluator, in terms of key concerns and associated challenges. We discuss some important background to OWL-based policy/data validation. Specifically, how it is possible to cast the goal of policy evaluation as either a problem of integrity constraint validation or a problem of object classification.

In the first paradigm, policies are represented as queries, and any objects returned by the query are cases that violate the integrity constraints. Policy conformance checking is therefore reduced to query answering. In the latter paradigm, policies are represented as OWL DL class axioms. Any objects that are classified as satisfying the description are either satisfying or not satisfying the policy, depending on whether the policy as encoded in positive or negative form. Policy conformance checking is therefore reduced to DL instance checking.

We present our own method following the second approach, and illustrate the process through an example. We discuss how the open-world assumption and the lack of a unique name assumption in the language semantics of OWL prevent us from using an OWL DL reasoner for RDF data validation, out of the box. We present a solution to overcome this challenge and combine the best from the worlds of OWA and CWA. The solution is based on generating local closure axioms covering those properties of governed resources which are important from a policy evaluation perspective.

The approach that we implemented as a prototype allows a standard OWL-DL reasoner to operate in a closed-world setting and produce the desired inferences so as to successfully check conformance of governed resources to ecosystem governance policies.

7.1.1 Governance from the policy evaluator's perspective

As discussed in chapter 3, the governance policy evaluator role is responsible for creating, maintaining, and providing a system that facilitates governance by carrying out policy evaluation. The system must check whether governed resources conform to the applicable policies or not. The inputs to the evaluation process are the policies and the descriptions of the governed resources.

To offer some examples, we will again refer to the ecosystem governance scenarios from chapter 3. The role of policy evaluator in scenario #1 is assumed by the quality review unit of NineLives. In scenario #2 the role belongs to the compliance management team at NineMed. In scenario #3 the role belongs to the CloudDev quality assurance staff, and in scenario #4 it is the auditors' firm Better, Saffe & Sawrie. In scenario #5 the role is assumed by Appregator, the cloud service broker.

Individual policy evaluators in the ecosystem are primarily concerned with effectively and efficiently managing the policy evaluation process internally and easily exchanging governed resource descriptions and policies with other ecosystem partners.

- Internal management objectives: To be able to freely make changes to how resource descriptions and policies are being processed, while containing these changes locally, i.e. without necessitating any corresponding changes to third-parties such as policy providers or data providers. To retain policy evaluation logic as generic and reusable as possible so as to avoid re-implementing (and risk introducing bugs when re-implementing) policy evaluation logic from scratch, for every different policy or different type of governed resource description.
- External communication objectives: To be able to understand governed resource description and policy definitions without needing to know or understand how these are processed or represented by policy providers and data providers.

7.1.2 Policy evaluation from the ecosystem's perspective

Observed from an ecosystem-wide perspective, policy evaluation engines exhibit the same three characteristics as discussed in section 6.1.2 for governed resources.

Heterogeneity:

- Policy evaluation engines implemented by different ecosystem partners will rely on much different technologies to process governance policies and resource descriptions.

Physical distribution:

- Software ecosystem partners are distributed, and so are the policy evaluators. Policy evaluation is a role that may be assumed by more than one partner in an ecosystem, in parallel.

Fragmented ownership/control:

- Policy evaluation mechanisms may evolve (be modified, optimised) independently of other governance stakeholders.

7.2 Query-based vs classification-based policy evaluation

Checking ontology-based descriptions of ecosystem resources against governance policies is a task which can generally be viewed as (at least) two different kinds of computational problem: a problem of *integrity constraint validation* on ontology objects, or a problem of *ontology object classification*. Depending on the adopted approach one must implement the appropriate strategy for defining policies and checking data against these policies.

When policy checking is cast as an integrity constraint validation problem the strategy is to define governance policies in the form of first-order logic queries. This can be done using an ontology-based query language such as SPARQL [113] or SQWRL [160]. Policy evaluation can then be reduced to query answering. If a query returns a non-empty result set, it means that the returned ontology objects violate the integrity constraints specified in the query, i.e. they do not conform to the policy.

Alternatively, policy evaluation can be approached as an object classification problem whereby governance policies are defined as Description Logic (DL) class axioms. This is the approach we have described in chapter 5. Under this approach, the task of policy evaluation can be reduced to instance checking with an OWL DL reasoner. Instance checking is a basic service provided by every DL reasoner [161] such as Pellet [127] or Hermit [128], to answer if a given individual is an instance of a specified class [162]. As we discuss next, due to certain characteristics of the OWL language this strategy requires an additional pre-processing step before instance checking is actually applied on any particular ontology object.

Literature on OWL and RDF data validation provides examples of both approaches in contexts bearing similarities -but also differences- to governance policy evaluation. For instance, the projects described in [163] and [164] adopt the first and second approach, respectively. Each approach has its own advantages and disadvantages in specific contexts, and this is very much dependent on the rest of the use cases, beyond governance, for which an organisation wants to use semantic data representation and ontology models. The return on investment that any organisation expects to see from the use of semantic technology, or any technology for that matter, is stronger when the technology is utilised to benefit more than one business processes; i.e. in this case, more than just governance policy enforcement. The choice of approach is therefore strongly linked to an organisation's other reasons for using semantic technology.

The policy evaluation component of the conceptual framework that we introduced in chapter 4 should be possible to realise with either of the two approaches. In this research, however, we have chosen to demonstrate the feasibility of the framework through the DL classification-based approach. The reason for preferring this approach is purely practical and has to do with the author's past experience and familiarity with developing software systems that utilise OWL-DL reasoning mechanisms.

7.3 Method for policy evaluation based on DL reasoning

The input to initiate the policy evaluation process is a pair of ontology-based governance policy definition and governed resource description. The method to create them has been discussed in chapter 5 and chapter 6, respectively.

7.3.1 Governance policy evaluation example

To illustrate the process of policy evaluation let us use an example from the previous chapter; the WSDL document describing the interface of a text translation web service used by apps on the CAST platform.

According to the WSDL document describing the service's interface, the service can be invoked at two different endpoints (URLs). The relevant section of the service's WSDL document is shown in Table 20 (section 6.3.1).

Extracting the information from the WSDL document and making it available as Linked Data would allow a policy evaluation engine to query the resource provider's RDF triple store for information about the service, using SPARQL (see Table 22), and receive the following RDF description as a response (Table 24).

```
@prefix : <http://kourtesis.net/phd/2016/examples#> .
@prefix gov: <http://ecosystem-governance.com/ontology#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@base <http://kourtesis.net/phd/2016/examples> .

:translationService  rdf:type  gov:ServiceInterface ;
:translationService  gov:contains  :endpoint-001 ;
:translationService  gov:contains  :endpoint-002 ;

:endpoint-001  rdf:type  gov:Endpoint ;
:endpoint-001  gov:hasAddress  "http://144.76.8.88"^^xsd:anyURI ;

:endpoint-002  rdf:type  gov:Endpoint ;
:endpoint-002  gov:hasAddress  "http://143.167.8.2"^^xsd:anyURI .
```

Table 24. Description of translation service interface (Turtle syntax)

The relevant artefact validation policy from the CAST platform dataset states that the interface specification of every external web service used by apps on the CAST platform should specify two unique endpoint URLs (primary and backup endpoints as failover).

Expressed as a DL axiom and represented in OWL Manchester Syntax, the policy is formulated as shown in Table 25.

```
Class: ValidServiceInterface

    EquivalentTo:
        ServiceInterface
         and (contains exactly 2 Endpoint)

    SubClassOf:
        ServiceInterface
```

Table 25. Definition of ValidServiceInterface policy (Manchester syntax)

For completeness, Table 26 shows the `ServiceInterface` artefact class which is referenced by the `ValidServiceInterface` policy definition. `ValidServiceInterface` is defined as a disjoint union of `InvalidServiceInterface` and `ValidServiceInterface`. As mentioned in the method description section of chapter 5, the disjoint union axiom in the class description will force a DL reasoner to deduce than any instance of a service interface artefact which cannot be classified under `ValidServiceInterface` is necessarily an `InvalidServiceInterface`.

```
Class: ServiceInterface

    SubClassOf:
        ServiceArtefact

    DisjointUnionOf:
        InvalidServiceInterface, ValidServiceInterface
```

Table 26. Definition of ServiceInterface artefact class

Once the policy evaluation engine obtains the resource description shown in Table 24 and the policy definition shown in Table 25, the objective is to check if the data satisfies the definition. This can be done by invoking the instance checking function of an OWL DL reasoner such as Pellet [127] or Hermit [128]. To enable instance checking with an OWL-DL reasoner, the policy evaluation engine will first construct an OWL individual to represent the described governed resource (Table 27), by importing the RDF description shown earlier in Table 24.

```
Individual: translationService

    Types:
        ServiceInterface

    Facts:
        contains  endpoint-001,
        contains  endpoint-002
```

Table 27. Description of translation service interface as OWL individual (Manchester syntax)

For completeness, Table 28 shows the encoding for OWL individuals `endpoint-001` and `endpoint-002`, which are referenced from within the description of the `translationService` individual.

```
Individual: endpoint-001

  Types:
      Endpoint

  Facts:
      hasAddress  "http://144.76.8.88"^^xsd:anyURI


Individual: endpoint-002

  Types:
      Endpoint

  Facts:
      hasAddress  "http://143.167.8.2"^^xsd:anyURI
```

Table 28. Description of translation service endpoint-001 and endpoint-002 as OWL individuals

Those three individuals (`translationService`, `endpoint-001`, `endpoint-002`) can be in-memory objects and do not need to be permanently stored. They can be generated on-demanded and discarded as soon as the instance checking function of the reasoner terminates.

By this point, the policy evaluation engine has provided the DL reasoner with an in-memory representation of the policy (a defined OWL class), and an in-memory representation of the instance data (OWL individuals), so the DL reasoner can proceed with instance checking to determine if the object of interest (`translationService`) can be classified under the class of interest, i.e. `ValidServiceInterface` (or its mutually disjoint counterpart, the `InvalidServiceInterface` class).

The object `translationService` is of type `ServiceInterface` and contains two objects of type `Endpoint`, each pointing to a different URL. Therefore, at first sight, it may intuitively appear as if the description of the object satisfies the definition of policy class `ValidServiceInterface`, and that the DL reasoner should infer that `translationService` is of type `ValidServiceInterface`. However, this is not the case. The reasons are explained in the following section.

7.3.2 Open-world assumption and unique name assumption

OWL as a language has two characteristics that militate against utilising an OWL DL reasoner for direct applications in data validation, as in the example above. These characteristics are the presence of an open-world assumption (OWA), and the absence of a unique name assumption (UNA) [162] in the language's semantics.

When drawing conclusions from the information available, knowledge representation systems that adopt the OWA will assume that, by default, the information held in a knowledge base is incomplete. They assume there could be more objects and more relationships between them than those which happen to be known to a knowledge base at the given time, and this information may become part of the knowledge base in the future. A reasoning engine that operates under an OWA will limit itself to drawing conclusions only if these follow from the facts already available in the knowledge base at the time of reasoning, without making any assumptions about the completeness of the information held. If something is not known for a fact to be true or false, its truth value could be either true or false, and the system will refrain from drawing any conclusion. The OWA is closely related to the monotonic nature of classic first-order logic: adding new information can never falsify a previously drawn conclusion [165].

By contrast, knowledge representation systems adopting a closed-world assumption (CWA) will draw conclusions assuming that the information held in a knowledge base is always complete. They assume there are no more objects and relationships between them other than those already known. If something is not known, its truth value is assumed to be false. The CWA is adopted in query-answering over relational databases and in Datalog-related logics. Datalog-based systems, such as Prolog programs, support non-monotonic reasoning and defeasible inference. That is, inference where reasoners draw conclusions tentatively, reserving the right to retract them in the light of further information [166].

Another semantic property that is closely related to the Datalog/relational paradigm is that names will uniquely identify and distinguish the objects in the domain. A resource can never be identified and referred to by two different names. This is known as the unique name assumption (UNA) and it is a common practice in non-monotonic reasoning [165]. However, it is not the assumption made in monotonic logic languages like OWL, where resources which are identified by different URIs are not assumed to be different objects in the domain, unless explicitly stated otherwise. Consistent with the openness and heterogeneity of the web, in OWL, there is nothing preventing a resource from being identified by several different names, by different entities.

As noted by Bergman [119], operating under a CWA (and UNA) is a “poor choice” when attempting to combine information from multiple sources, to deal with uncertainty or incompleteness in the world, or to try to integrate internal, proprietary information with external data. Situations where a default CWA is more convenient and appropriate than OWA are “database-like applications” [167], where knowledge is mostly managed in a local scope. Data validation is one such application, where what is effectively desirable is to encode and check the integrity constraints that must be satisfied by instance data.

Overall, there are several use cases identified by the Semantic Web community [168], [169],[170] where it is desirable to adopt the OWA without the UNA for parts of the

domain where knowledge is incomplete, and simultaneously, use the CWA with the UNA otherwise [171].

Fortunately, there are ways to combine the best of the two approaches: the open world reasoning of OWL with closed world constraint validation.

7.3.3 Local closure axiom generation algorithm

According to the standard semantics of OWL, an OWL DL reasoner will never infer that some statement is false simply because there is no known evidence to support the truth of that statement. And it will never infer that two objects are distinct because they are identified by different names.

A solution in order to guide the DL reasoner to draw the desired inferences is to enable some form of local closed world reasoning. That is, to close the world relative only to the descriptions of governed resources we are interested in checking, while leaving the rest of the knowledge base (KB) to be processed under the standard OWA. This can be accomplished by adding additional assertions regarding the object of interest, so as to state that all the information relevant to that object is known.

Going back to the translation service example, let us see how an OWL DL reasoner would fail to produce the desired inferences, and how we can approximate local closed world reasoning.

Let the policy presented in Table 25 earlier in this chapter be represented by an equivalence class axiom as in (1):

$$\texttt{ValidServiceInterface} \equiv \texttt{ServiceInterface} \sqcap (=2\ \texttt{contains.Endpoint}) \quad (1)$$

Let knowledge base $\mathcal{K}$ contain the ontology instance data as in (2):

```
K = {
ServiceInterface(translationService),
Endpoint(endpoint-001),
Endpoint(endpoint-002),
contains(translationService, endpoint-001),
contains(translationService, endpoint-002)  }
```
(2)

Given the above pair of governance policy and governed resource description, we would like the reasoner to infer `ValidServiceInterface(translationService)`, which is a way of saying that individual `translationService` belongs to the class of valid service interfaces. However, without any UNA, the combination of (1) and (2) does not entail this inference. Despite the fact that `translationService` is known to contain the endpoints `endpoint-001` and `endpoint-002`, there is nothing to preclude that `endpoint-001` and `endpoint-002` is not in fact the same individual.

To compensate for the lack of UNA we can extend $\mathcal{K}$ by asserting explicitly that `endpoint-001` and `endpoint-002` are different individuals (3):

`endpoint-001 ≠ endpoint-002` (3)

However, even with this addition it is still not yet possible to have `translationService` classified under the anonymous class of things that contain exactly two endpoints `(=2 contains.Endpoint)`. Due to OWL's OWA, the cardinality restriction in the class expression can be satisfied only if we have explicit knowledge that `endpoint-001` and `endpoint-002` are in fact the only objects related to `translationService` along the `contains` property (otherwise, `translationService` could be related to more, as yet unseen objects along this property). The way to achieve this is by extending $\mathcal{K}$ with an anonymous type assertion as in (4):

`(=2 contains.T) (translationService)` (4)

The assertion states that the number of objects to which `translationService` is related along `contains` property is exactly 2. The addition of (3) and (4) to the KB is a way of closing the world relative to a part of the KB (i.e. relative to `translationService` only) and in isolation from other ontology individuals.

It is worth noting that the specific assertions (3) and (4) are not the only ways to achieve the desired closure. There are other modelling constructs in OWL that could be used to achieve the desired effect, such as enclosing the full list of objects that `translationService` is related to along the `contains` property in a universal quantification axiom like `(∀contains.{endpoint-001, endpoint-002}) (translationService)`, or using the `hasAddress` data property as a Key for all `Endpoint` objects, which would allow the reasoner to distinguish between different service endpoints if their URIs are different.

These special-purpose assertions do not need to be custom-coded or predefined in templates. They can be dynamically generated by a generic mechanism, as a pre-processing step within the policy checking engine. Moreover, the addition of such assertions does not need to be permanent – they can be discarded as soon as conformance checking for `translationService` has been completed.

This is achieved by our local closure generation algorithm which examines the equivalence class axiom representing a policy of interest, determines which (asserted or inferred) properties are relevant for classification, constructs anonymous type assertions with the exact known objects or literals per each property of importance, and adds those to the object to be checked.

The steps in the local closure generation algorithm are as shown in Table 29:

```
01   Get URI of the governed resource (OWL individual) to check
02   Get URI of the policy definition (OWL class) to check against

03   Check well-formedness of resource description against pattern

        // Well-formed individuals have exactly one asserted type class

04      If individual is not well-formed, exit with error

05   C heck well-formedness of policy description against pattern

        // Well-formed policies are pairs of positive/negative policy
        // descriptions under parent class with DisjointUnion axiom.
        // They also follow a specific naming pattern:
        // ValidABC/InvalidABC for artefact validation
        // PromotableABC/NonPromotableABC for lifecycle management

06      If class is not well-formed, exit with error

07   Discover the salient object and data properties relative to the
     policy

08      Analyze OWL Object Property restriction expressions in policy

        // MinCardinality, MaxCardinality, ExactCardinality,
        // AllValuesFrom, SomeValuesFrom, HasValue, HasSelf

09      Analyze OWL Data Property restriction expressions in policy

        // MinCardinality, MaxCardinality, ExactCardinality,
        // AllValuesFrom, SomeValuesFrom, HasValue

10   Generate closures for salient policy properties

11      If salient property is OWL Object property

12         Get a reference to the predicate (object property)
13         Get a reference to the subject (individual)
14         Get all objects to which the subject is related
15         Create ObjectAllValuesFrom expression for property/objects
           // Example: contains only {endpoint-001, endpoint-002}

16      Add the closure axiom to the KB

17      If salient property is OWL Data property

18         Get a reference to the predicate (data property)
19         Get a reference to the subject (individual)
20         Get all literals to which the subject is related
21         Create DataAllValuesFrom expression for property and filler
           // Example: hasPrice only ({10})

22         Add the closure axiom to the KB

23   Load KB (ontology, imports closure, closure axioms) onto DL reasoner

24   Run KB consistency check with DL reasoner

25      If KB is consistent
```

```
26          Check if individual is instance of positive or negative policy
            class
            // Example: Type ValidServiceInterface or Type
            InvalidServiceInterface

27          Return instance check result

28      Exit
```

Table 29. Abstract description of local closure generation algorithm

Table 30 and Table 31 below show the encoding for `translationService` and individuals `endpoint-001` and `endpoint-002`, with the addition on the closure axiom assertions from (3) and (4) above.

```
Individual: translationService

    Types:
        ServiceInterface,
        contains exactly 2 Endpoint

    Facts:
        contains  endpoint-001,
        contains  endpoint-002
```

Table 30. Description of translation service interface as OWL individual (Manchester syntax)

```
  Individual: endpoint-001

    Types:
        Endpoint

    Facts:
        hasAddress  "http://144.76.8.88"^^xsd:anyURI

    DifferentFrom:
        endpoint-001


  Individual: endpoint-002

    Types:
        Endpoint

    Facts:
        hasAddress  "http://143.167.8.2"^^xsd:anyURI

    DifferentFrom:
        endpoint-002
```

Table 31. Description of translation service endpoint-001 and endpoint-002 as OWL individuals

Using the local closure axiom generation algorithm that we created, any OWL DL reasoner presented with the above instance data and the definition of `ValidServiceInterface` policy in Table 25 would infer that `translationService` is of type `ValidServiceInterface`, and therefore establish policy conformance.

7.3.4 Related work

The formulation of constraints and the automatic validation of data according to these constraints is a much sought-after feature for RDF/OWL applications [172], and is growing in importance.

Closely related work has been carried out on theory and applications for OWL and RDF data validation. Some works have focused on ways to extend the semantics of OWL to allow for integrity constraint validation. This typically involves the use of non-monotonic queries over the knowledge base, and the language used is typically SPARQL. Other works have focused on ways to implement local closed world reasoning over the knowledge base without introducing alternative semantics for OWL.

Motik, Horrocks and Sattler [170] have proposed an extension of OWL semantics with Integrity Constraints (IC) similar to those found in relational databases. Their approach allows a subset of tBox axioms to be designated as ICs, which are interpreted in the spirit of relational database constraints during aBox reasoning. Bringing this in the context of policy-based governance, the tBox axioms would represent the policies, while the aBox would contain the instance data for the governed resource descriptions to be checked.

Tao, Sirin, Bao, and McGuinness [163] also describe an alternative IC semantics for OWL, based on CWA and weak UNA. Their approach allows developers to augment OWL ontologies with IC axioms and combine open world reasoning with closed world constraint validation. They also show that, under certain conditions, IC validation can be reduced to query answering through SPARQL queries which are automatically generated from OWL DL class axioms.

SPARQL Inferencing Notation (SPIN) [173] has similar objectives. It was created to serve as a SPARQL-based rule and constraint language for the Semantic Web. It allows ontology class definitions to be linked to SPARQL queries in order to capture constraints and rules that formalise the expected behaviour of objects belonging to those classes. In practice, SPIN offers a way to do constraint checking with closed world semantics and raise inconsistency flags when the currently available information does not fit the specified integrity constraints.

A related tool implementation is presented by Rieckhof, Dibowski and Kabitzsch [174], who are interested in formal validation techniques for ontology-based electronic device descriptions. They describe the implementation of a validator that checks for

consistency, completeness and correctness in device descriptions using SPARQL queries.

Miksa, Sabina and Kasztelnik [164] present a prototype system for ontology-based modelling of network devices. Their motivation is to detect configuration errors and to propose combinations of compatible devices by means of instance checking and other DL reasoning services. They achieve this by implementing a method similar to our own in order to implement local closed world reasoning and thus be able to detect configuration errors, while keeping the rest of the KB "open" in order to properly reason about combinations of compatible network devices [175].

7.4 Summary

In this chapter we presented the third and final part of our implementation of the conceptual framework put forward in chapter 4. We discussed how to enable evaluation of the governance resource descriptions described in chapter 6 against the ontology-based governance policy definitions described in chapter 5.

The governance policy evaluator role is responsible for creating, maintaining, and providing a system that checks the conformance of governed resources to ecosystem policies. Providing this capability in the form of an automated policy evaluation engine is at the centre of every scalable policy enforcement infrastructure.

Observed from an ecosystem-wide perspective, policy evaluation engines exhibit the same three characteristics as discussed for policies and governed resources: they are physically distributed, owned and controlled by different ecosystem actors, and unless standardised and generalised through our proposed approach, they can be heterogeneous and incompatible between different ecosystem partners.

We presented a method for policy evaluation based on DL reasoning which allows distributed and independent policy evaluators to use a common policy conformance checking infrastructure. One that is generic enough to cover all the different types of governance policies and governance resource descriptions in the ecosystem.

We presented the background to our proposed method and explained the differences between query-based and classification-based evaluation approaches for ontology-based policies. Through an example from the CAST platform governance dataset, we illustrated how classification-based evaluation with a DL reasoner will stumble upon standard OWL semantics, and what are the effects of the open-world assumption and the lack of unique name assumption in OWL.

We then presented our implementation of an algorithm for local closure axiom generation that allows an open-world reasoning engine such as a standard OWL-DL reasoner to operate in a closed-world setting and produce the desired inferences so as to

successfully check conformance of governed resources to ecosystem governance policies.

The algorithm examines the equivalence class axiom representing a policy of interest, determines which properties are important for object classification purposes, constructs anonymous type assertions for the governed resource description with the exact known objects or literals per each property of importance, and adds those to the knowledge base before instance checking (classification) is performed.

The overall approach allows us to combine the best of open-world and closed-world reasoning approaches in a single framework: Open-world reasoning when using ontologies and Linked Data to integrate heterogeneous and incomplete information from different sources, inside and outside a software ecosystem; Closed-world reasoning where knowledge in a local scope can be considered complete, such as with governance policy evaluation.

The key takeaways from this chapter can be summarised as follows:

1. The function of the governance policy evaluator is to create, maintain and provide a system that facilitates governance by carrying out policy evaluation. The role of the system is to check whether a given description of a governed resource conforms to the applicable policies.
2. Checking ontology-based descriptions of resources against governance policies is a task which can be approached as two different kinds of computational problem: a problem of integrity constraint validation on ontology objects, or a problem of ontology object classification. The policy evaluation component of the PROBE framework should be possible to realise with either of the two approaches. In this research we have chosen to explore the DL classification-based approach.
3. OWL as a language has two characteristics that prevent us from directly utilising a DL reasoner's object classification service for data validation. These characteristics are the presence of an open-world assumption (OWA) and the absence of a unique name assumption (UNA) in the language's semantics. To overcome this obstacle we present an algorithm for local closure axiom generation. The algorithm closes the world relative only to the descriptions of governed resources we are interested in evaluating, while leaving the rest of the knowledge base to be processed under the standard OWA.
4. The algorithm works by automatically generating additional assertions regarding an object of interest, i.e. a governed resource description, so as to state that all the information relevant to that object is known, i.e. to "close" the world. The strength of the algorithm is in its generality and reusability. Assertions do not need to be custom-coded or predefined in templates but can

be dynamically generated as a pre-processing step within the policy evaluation engine.

5. Overall, our proposed method for evaluating governance policies combines the best of two worlds: the open world reasoning of OWL with closed world constraint validation. The approach relieves the ecosystem partners that function as policy evaluators from the need to develop and maintain a custom semantic policy evaluation engine. Instead, they can use a standard OWL DL reasoner in combination with our generic local closure axiom algorithm. The resulting policy evaluation mechanism is natively suited to support networked collaboration and is generic enough to cover all the different types of governance policies and governed resource descriptions in the ecosystem.

Chapter 8

Comparative case study

8 Comparative case study

8.1 Introduction

In the preceding chapters we analysed the key requirements of governance processes in cloud service ecosystems, we discussed the limitations in state of the art governance technology platforms and introduced a new conceptual framework for developing governance support systems. The PROBE framework utilises Linked Data principles and Semantic Web standards to achieve the desired evolutionary step in terms of enabling networked collaboration between governance process stakeholders and improving the operational efficiency of governance processes.

To demonstrate the feasibility of the PROBE framework we showed how its components can be concretely realised in order to define policies, to describe governed resources and to evaluate governance policies against resource descriptions, drawing on governance policy examples from research project CAST.

In this chapter we demonstrate the usefulness of the PROBE framework. This constitutes the second component of our research validation strategy.

To demonstrate the usefulness of the PROBE framework we compare a PROBE-based governance support system with the governance support system that was created in the scope of research project CAST. The two systems are compared in two different ways. The first comparison is implicit, in that it happens by describing how each system supports policy definition, data extraction and policy evaluation, and what the implications of each design approach are. The second comparison is explicit, in that we compare one system vis-à-vis the other system with the help of change scenarios that allow us to analyse how each system behaves under interesting conditions. The systems are examined from the perspective of the different roles involved in the ecosystem governance process and change-scenarios help to evaluate how each system supports evolvability and manageability of the governance process.

The chapter starts with an introduction to the CAST project and the cloud application platform that the project consortium created. We discuss the governance requirements that emerge for this particular type of cloud platform and present several governance policy examples. Following, we introduce the governance support system that was developed in the CAST project to address those requirements and then describe an alternative approach to supporting policy-based governance for the CAST platform based on the PROBE framework. For both design approaches we explain how policy definition, data extraction, and policy evaluation are supported.

Finally, we evaluate the two design approaches based on the requirements analysed in section 3.4. We examine the two approaches from the perspective of the different roles

involved in the ecosystem governance process: policy provider, data provider, and policy evaluator, using scenario-based comparison. Change-scenarios are employed to assess the two design approaches in terms of how they support evolvability and manageability of governance processes.

8.2 CAST project

Project CAST was a joint industry-academia research project that run from 2009 to 2012 and developed technology to support co-development of cloud-based business software solutions by ecosystems of software companies [3],[22]. In the scope of project CAST the author of this dissertation analysed a set of 37 governance policies relating to lifecycle management and artefact validation for the cloud application platform created by the project and its governance support system. The author also led the technical team that developed a governance support system based on an open-source registry and repository platform to enforce these policies at run-time [121],[122]. Through this exercise the author gained valuable insights into the strengths and limitations of current approaches to policy-based governance as implemented by state-of-the-art governance support systems.

8.2.1 Background

The CAST project was set up to investigate the engineering challenges associated with creating a PaaS platform that enables ecosystem-oriented development of business software solutions [22],[3]. The project did not set out to create yet another PaaS platform for generic web application development, but one focusing on business applications co-developed by cloud service ecosystems. The models and technology developed as part of the project have found their way to commercial products by CAS Software AG and follow-on research programmes such as Broker@Cloud [11], [10], [12],[13].

One of the project's main outcomes was the CAST platform: a PaaS software infrastructure comprising an SDK for the development of applications, an application runtime environment and a set of supporting platform management tools. The most distinctive characteristic of the CAST platform is that its design is oriented towards creating network effects [176], by fostering the emergence of an ecosystem of business software creators around the PaaS. To promote this objective, the platform allows developers to create "solutions" by combining reusable prebuilt components (referred to as "apps") which may be offered not only by the platform provider –as commonly happens in PaaS platforms, but also created and offered by independent third-parties.

Enabling developers to construct applications this way—i.e. developing software on top of a PaaS platform through the reuse of building blocks provided by third-parties within the platform's ecosystem, is a major force in the market of cloud application

platforms [177]. This trend promotes reusability, but also creates a need for much more advanced platform mechanisms for quality assurance. The openness and complexity of the environment that emerges makes stability and reliability much harder to guarantee, and calls for a rigorous approach to platform governance. This was an important requirement that the CAST project undertook to research.

Before proceeding to discussing specific requirements with respect to governance in the context of the CAST platform and how these requirements were addressed, the next sections provide a brief introduction to the main concepts and terminology relating to the CAST platform.

8.2.2 CAST platform concepts and terminology

8.2.2.1 Development constructs

There are three fundamental constructs that shape enterprise applications developed based on the CAST PaaS platform: solutions, apps and services.

- *CAST platform solutions.* A solution is defined as a complete enterprise software application that targets a specific application domain or market niche (e.g. customer relationship management for French insurance companies, or event management for German exhibition centres). It is deployed on the CAST platform and made available to end-users as on-demand software (SaaS). Unlike a typical web application, a solution is not manifest as executable artefacts – there are no code binaries that form a solution, just metadata. This is because a solution is effectively a logical bundle of finer-grained components which provide the actual functionality.
- *CAST platform apps.* The finer-grained components that solutions are composed of are called apps. Each app within a solution provides a highly-specialised function. An app in CAST can be characterised as either data-centric or process-centric. A data-centric app provides the implementation for creating, viewing, editing and storing a custom-built data object (for example, an employee's record, or a project's timesheet). A process-driven app provides the implementation for supporting an end-user in carrying out a sequence of tasks (for instance, supporting a sales employee for mass-importing customer addresses from a spreadsheet file). Developing an app involves coding against platform APIs which may span three different platform runtime layers. An app may define new data object types on the data layer, new business operations on the business logic layer, and new user interface elements on the presentation layer. An app's behaviour can be extended by creating app extensions, which interface with the app at designated extension points. An app extension is therefore not a standalone component, but functions as a plug-in to one or more different apps.

- *External services.* Apps and app extensions may rely on external services to deliver part of their functionality. By external services we refer to systems that are deployed and executing outside the platform and are accessible over the Web, through a programmatic interface (i.e. REST or SOAP Web services). The ability to use Web services enables the developers of solutions to leverage already existing (and tested) solutions for particular specialised tasks within their apps. For example, an app or app extension for contact management could invoke an external service to perform email address validation for a particular contact, or to obtain the latest mentions on social media for a contact's company. Even more importantly, the use of Web services in conjunction with apps makes it possible to integrate solutions which are deployed on the platform with external organisations and service providers, as well as legacy systems.

8.2.2.2 Ecosystem co-development on the CAST platform

The development constructs introduced above represent a generic model that could be applicable to a wide range of cloud application platforms. But how do these constructs map to specific entities/actors in the cloud service ecosystem that the CAST platform facilitates? Who creates and who extends those constructs in the context of enterprise application co-development?

Figure 17 illustrates an example of possible associations/dependencies between platform constructs (solutions, apps and services) and organisational boundaries of different ecosystem partners (dashed outline).

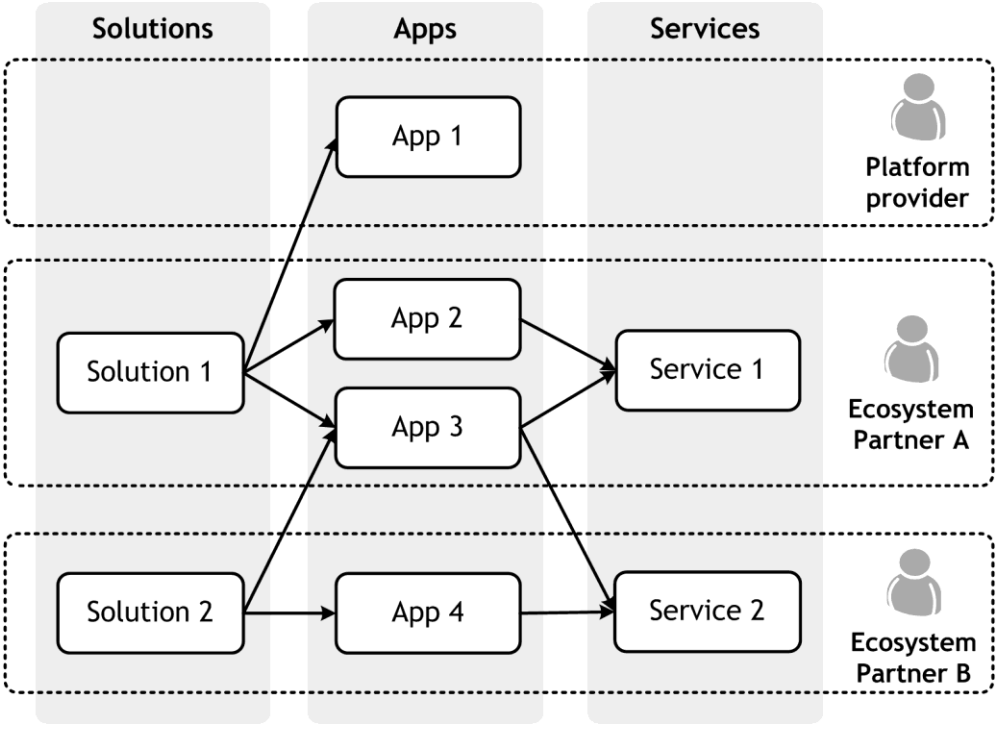

Figure 17. Example mapping of platform constructs to ecosystem partners

Apps and app extensions may be built both by the platform provider and by third-parties (ecosystem partners A and B). In order to help developers to bootstrap their work, the platform provider may build a number of apps that target functionality that is rather common in business applications, such as document management (app 1).

Any partner that needs to use a built-in app is allowed to configure it for the needs of a particular solution (solution 1). Alternatively, apps and app extensions can be developed from the ground-up by different ecosystem partners (apps 2, 3, 4). Optionally, those apps can depend on external services (service 1, 2) which may not necessarily be owned by the same ecosystem partner (app 3, service 2). In any case, as soon as a third-party app, app extension, or external service is onboarded to the platform it can be made available for other partners to reuse in developing their own software.

Composing built-in and third-party apps into solutions is the responsibility of ecosystem partners (solution 1, 2). In creating a solution package, ecosystem partners are also specifying how the appearance and behaviour of the included apps should be customised (at run-time) for the particular solution at hand. This is done by defining solution-specific constraints on the apps.

Since an app can be part of more than one solution (app 3), different constraints can be applied for a particular app depending on the execution context. For example, data validation rules for fields such as a postal code or a vehicle license plate can be customised at app run-time quite differently, depending on the relevant country the solution should support.

8.3 Policy-based governance of CAST platform

8.3.1 Governance requirements

As the usage of any CAST platform instance expands over time, increasing numbers of solutions, apps and services will be deployed and updated on a frequent basis. To ensure that appropriate operation conditions are maintained at all times, it is imperative to implement mechanisms for effective, centrally-exercised management and quality control over all of the platform's solutions, apps and services.

Essential to achieve this is to have a central governance support system in which all entities and artefacts that are intended for deployment to the platform's runtime environment can be stored, organised, checked for conformance against platform policies, and managed throughout their lifecycle.

The governance support system needs to facilitate a systematic and disciplined process for the deployment of solutions, apps and services to the platform, such that potential

problems can be detected and addressed before posing a threat to the platform's stability and performance.

The CAST project identified five core governance functionalities as most critical:

- Cataloguing and storage: Solutions, apps, and services need to be catalogued, and their associated artefacts stored in a central location.
- Artefact validation: Managed entities (solutions, apps, and services) and their associated artefacts need to be checked for conformance to platform policies.
- Lifecycle management: The evolution of managed entities must follow an explicitly defined lifecycle model consisting of specific states and transitions, where transitions are guarded by preconditions.
- Dependency tracking and impact analysis: The dependencies among solutions, apps, and services need to be tracked to allow for impact analysis.
- External service monitoring: External (Web) services on which apps may depend must be monitored to ensure availability and performance.

8.3.2 Stakeholders in the governance process

In section 3.3 we discussed the three fundamental roles which interact in the scope of any cloud service ecosystem governance process: i) policy provider, ii) data (resource description) provider, and iii) policy evaluator.

Those three distinct roles are also present in the setting of a CAST platform instance.

- **Policy provider:** This role can be assumed by both the company that provides/operates the CAST PaaS platform, as well as the Independent Software Vendors (ISVs) who are ecosystem partners. For instance, the platform operator sets out a range of policies to govern the structure of the solutions and apps that are deployed onto the platform's runtime environment, as well as policies governing how the platform can interface with external services. On the other hand, ecosystem partners set policies/constraints on the third-party apps or services used by their solutions, such as performance/availability or cost limits.
- **Data provider:** Similarly, the role of the data provider can also be assumed by both the CAST platform operator as well as by ecosystem partners. For instance, the platform operator needs to deliver resource descriptions relating to the apps and services that are deployed onto the platform, so that this can be checked against relevant policies set out by ecosystem partners or the platform operator itself. Ecosystem partners also need to deliver resource descriptions relating to the apps and solutions they would like to deploy on the platform, so that they can be checked for conformance.

- **Policy evaluator:** Unlike the other two roles, the role of the policy evaluator is the responsibility of the CAST platform provider alone. As the operator of the platform, it is positioned at the centre of the ecosystem and needs to ensure that all the contributions by ecosystem partners can be smoothly integrated into a common operational environment. This requires evaluating all policies originating from ecosystem partners or by the platform provider itself, and governance aspects such as how to structure solutions or how to use external services from within an app, how to describe the pricing model of an app or how to use visual assets in the description of a solution.

8.3.3 Governance policy examples

To illustrate the type of governance policies relevant to CAST we list here a few examples. Specifically, we provide two examples of policies relevant to each different type of CAST platform software unit (solutions, apps, services).

8.3.3.1 Solution policy examples

Solution pricing.xml policy

A solution pricing definition artefact is an XML file which is created by the ecosystem partner who develops a solution and is submitted to the platform along with the rest of the definition/configuration artefacts that comprise a solution. It provides a definition of the solution's pricing model in terms of its subscription modality (for instance, charging a company that uses the solution per user seat - per month) and the billing amount per subscription option.

Example rules:

- Instance document is valid with respect to applicable XSD schema (pricing.xsd)
- Subscription modality constraints. For example, the modalities specified are either per user - per month, or per tenant - per month.
- Billing amount constraints: For example, the billing amount does not exceed 20 EUR, if modality is per user - per month, or 100 EUR, if modality is per tenant - per month.

Solution collection policies

A solution collection is the set of artefacts and metadata that comprise a solution and is provided by the ecosystem partner who creates the solution. The platform operator needs to validate that the solution collection contains a minimum set of artefacts and metadata. This minimum set is not fixed but can change depending on the lifecycle state of the solution –more constraints are added as a solution advances through its lifecycle.

The different states that a solution (or any other governed software unit in CAST) can sequentially move through during its lifespan are as follows:

1. Development
2. Testing
3. Review
4. Beta
5. Production
6. Deprecated
7. End of Life

Example rules:

Before allowing a transition from Development to Testing

- The collection's media type is "application/vnd.cast.sln"
- There exists exactly one valid solution.xml file
- All apps that this solution depends on, or recommends, are in the state of testing, beta, or production
- Any other artefact in the collection (e.g. any additional property file) is valid with respect to the policies applicable

Before allowing a transition from Testing to Review

- There exists a non-empty text description of the solution
- There exists a valid pricing specification file
- There exists a valid license file
- There exists a valid provider details file
- All apps that this solution depends on, or recommends, are in the state of beta, or production

Before allowing a transition from Review to Beta

- There exists a valid review report with positive outcome (approval=true)

Before allowing a transition from Beta to Production

- All apps that this solution depends on, or recommends, are in the state of production

8.3.3.2 App policy examples

App image files policy

An app image file is a visual asset that is used in the platform as part of the app description. It may be the app icon/thumbnail or a screenshot of the app in operation (e.g. to be used inside the CAST platform operator's app store). The app image artefacts are provided by whoever is the app creator (the CAST platform operator or an ecosystem partner). There exists a generic set of constraints that applies to all images, and a set of additional constraints which applies to app icons alone.

Example rules:

- The filetype is either jpg or png
- The maximum filesize is 100 KB for app icons and 1024 KB for all other images
- The maximum height is 150 pixels for app icons and 600 pixels for all other images
- The maximum width is 150 pixels for app icons and 800 pixels for all other images

App localisation files policy

An app localisation file is a .property artefact submitted to the platform by the app creator (the CAST platform operator itself or an ecosystem partner). It contains a list of key-value pairs that allow labels inside apps to be presented in different languages at run-time.

Example rules:

- The file is not empty
- Each defined key has some corresponding non empty value

8.3.3.3 Service policy examples

Service interface policy

A service interface definition is an XML artefact submitted to the platform by the service provider or someone who has created a wrapper for a third-party web service. It provides a machine-readable description of how the service interface can be invoked, what parameters it expects, and what data structures it returns, based on WSDL v2 as the definition language.

Example rules:

- Interface description is a valid WSDL v2 document
- Two service endpoints specified for redundancy purposes (primary/backup server)

Service level agreement policy

A service level agreement definition is an XML artefact which, similarly to the service interface definition, is submitted to the platform by the service provider or whoever else created a wrapper for a third-party web service (the CAST platform operator or an ecosystem partner). It provides some baseline metrics regarding service quality, based on WSLA as the definition language.

Example rules:

- The instance document is valid with respect to XSD schema (WSLA)
- The specified availability (uptime ratio) is no less than 98%
- The specified maximum response time is no more than 600ms

8.4 Description of the solution as developed in CAST

8.4.1 Overview - CAST platform registry & repository system

The approach that was taken in the CAST project to address the governance requirements outlined in section 8.3.1 was to develop a special-purpose registry & repository system that complements the CAST platform runtime environment.

The system was developed as a set of extensions and customisations on top of the open-source WSO2 Governance Registry platform [178], which offers part of the necessary infrastructure as well as generic interfaces and extension points that allow default functionality to be extended [121], [122].

In the following we explain the five main types of governance functions offered by the CAST Registry & Repository system.

Cataloguing and storage

The basis for all governance functions is the ability to catalogue solutions, apps, and services, and storing their associated artefacts. Each of the managed software unit in the platform comprises several artefacts (files) of different types, which need to be stored and linked to their associated unit. For example, each solution comprises a main descriptor file (manifest), a set of localisation files (property files), a specification file for data constraints that the solution places on apps, a pricing specification file, a license file, images, etc.

Central cataloguing and storage creates an authoritative system of record in which all data and metadata about the platform's assets are collected. Apart from the platform administrators, the cataloguing function allows platform users (i.e. software developers) to keep track of their portfolio of solutions, apps, and services, as well as those of other developers that are being reused in their work. Information about managed entities is also accessible to the platform's runtime environment through a programmatic API, allowing operations such as automated deployment of applications.

Artefact validation

The artefact validation function comprises a variety of automated processes to perform quality control on artefacts associated with managed CAST platform entities. This is done by checking the conformance or artefacts to pre-specified platform policies, which can be seen as a kind of integrity constraints.

We have already mentioned the policy example stating that every WSDL document that describes the interface of an external Web service should contain two endpoint URLs, pointing to two different servers on which the service is deployed. This provides a failover solution in case the primary server that hosts an external service is unavailable.

Validation of artefacts against such policies is triggered automatically whenever artefacts of interest are created or modified.

Lifecycle management

Lifecycle management refers to the ability of managing the state of governed software unit throughout their lifespan. Each managed CAST platform unit (solutions, apps, and services) follows a lifecycle model which is defined in terms of (i) states, (ii) transitions between states, (iii) pre-conditions to check before allowing a transition to a new state, and (iv) post-conditions to enforce upon exiting a state.

As already mentioned above, all of the CAST platform entities go through the states of: development, testing, review, beta, production, deprecation, and end-of-life. Pre-conditions and post-conditions differ between lifecycle model definitions for solutions, apps and services.

For example, as mentioned above, a pre-condition to automatically check before allowing a solution to proceed to the beta state is the availability of a certification report (an XML file provided during the review state by a member of the QA team), which should contain a positive evaluation for the app in question. An example of a post-condition to be enforced upon exiting the beta state would be to send a notification to some designated contact person, or to initiate a process of automated deployment to the production environment.

Dependency tracking and impact analysis

The dependency tracking function is concerned with specifying dependency associations among managed entities. That is, specifying dependencies between a solution and the apps it comprises, or an app and the external services that it may consume. This information is vital in order to prevent failures and to preserve the integrity of the platform. If the software unit at the right hand side of a dependency association is significantly modified or is removed from the platform (e.g. by being moved to the "end of life" state), the unit on the left hand side of the relation runs the risk of failing during runtime.

Keeping track of dependency information enables the platform to create warnings and/or deny unsafe user actions, such as attempting to remove a software unit to which there exists a direct or indirect dependency.

External service monitoring

As already mentioned, the apps that are offered through the CAST cloud application platform may rely on external Web services to deliver some of their functionality. This means that external services are essential parts of the ecosystem that is created around the platform, but at the same time, they lie beyond the control of the platform operator, since they are physically located and executed outside the platform's boundaries. If a service becomes unavailable or its performance is severely degraded, it could have a dramatic impact on a number of apps (and by extension on a number of solutions) which may be using it. To prevent potential problems of this nature it is critical to employ monitoring for all external services.

8.4.2 Policy definition

In this section and for the rest of this chapter we will focus on the two principal governance functions implemented by the CAST platform governance support system:

- Artefact validation policies
- Lifecycle management policies

We will use some examples of policies from section 8.3.3 to illustrate how policy definition was made possible with the CAST platform governance support system.

8.4.2.1 Definition of artefact validation policies

To illustrate how artefact validation policies can be defined with CAST, let us start with the example of validating the properties of visual assets that are part of an app description.

As seen in section 8.3.3, an app creator can provide an app icon/thumbnail or a screenshot of the app in operation to be used inside the CAST platform app store. To

maintain consistency, the properties of those files must be checked, and this task can be automated.

In the case of app image artefacts, policy definition is a four step process:

1. Creating an artefact validation policy rules file in XML format.
2. Implementing a new Java object converter to create an object representation of the policy rules file at runtime.
3. Implementing a new Java policy validator to validate the resource against the policy rules at runtime.
4. Packaging the Java components as JAR archives, deploying them to the policy evaluator's execution environment, and restarting the Registry & Repository server (Java servlet container) to complete the deployment.

Table 32 below presents the contents of the policy rules file. Constraints are placed on file type (valid extensions), and maximum the values allowed for width, height and file size.

```
<?xml version="1.0" encoding="UTF-8"?>
<policy>
    <name>Images</name>
    <isEnabled>true</isEnabled>

    <rule name="width">800</rule>
    <rule name="height">600</rule>
    <rule name="size">1024</rule>
    <rule name="extensions">jpg,png</rule>
</policy>
```

Table 32. Images.xml

Table 33 provides a snippet of code from the Java object converter that creates an object representation of the policy rules file. It reads in the XML object and expects to find values for the constraints placed specifically on extensions, width, height, and file size. Notice the tight coupling between the XML policy rules file and this object converter. If the structure of the first changes the code in the latter needs to be modified and redeployed to the CAST platform.

```
package org.seerc.cast.regrep.validators.app;

[...]
public class ImagePolicy {

    private static final Log log = LogFactory.getLog(ImagePolicy.class);

    private int width;
    private int height;
    private int size;
```

```
    private boolean isEnabled;
    private String extensions[];

    public ImagePolicy(String policyPath) throws RegistryException {

[...]
        try {
            log.debug("Reading Image Policy resource: " + policyPath);

            OMElement element;
            AXIOMXPath xpath;

            log.debug("Setting Policy isEnabled: " + policyPath);
            xpath = new AXIOMXPath("/policy/isEnabled");
            element = (OMElement) xpath.selectSingleNode(xmlDoc);
            this.isEnabled = Boolean.parseBoolean(element.getText());

            log.debug("Setting Policy rule [WIDTH]: " + policyPath);
            xpath = new AXIOMXPath("rule[@name='width']");
            element = (OMElement) xpath.selectSingleNode(xmlDoc);
            this.width = Integer.parseInt(element.getText());

            log.debug("Setting Policy rule [HEIGTH]: " + policyPath);
            xpath = new AXIOMXPath("rule[@name='height']");
            element = (OMElement) xpath.selectSingleNode(xmlDoc);
            this.height = Integer.parseInt(element.getText());

            log.debug("Setting Policy rule [SIZE]: " + policyPath);
            xpath = new AXIOMXPath("rule[@name='size']");
            element = (OMElement) xpath.selectSingleNode(xmlDoc);
            this.size = Integer.parseInt(element.getText());

            log.debug("Setting Policy rule [EXTENSIONS]: " +
policyPath);
            xpath = new AXIOMXPath("rule[@name='extensions']");
            element = (OMElement) xpath.selectSingleNode(xmlDoc);
            String extensionsStr = element.getText();

            if (extensionsStr == null) {
                log.debug("Policy Rule [EXTENSIONS] was not set: "
                        + policyPath);
            } else {
                this.extensions = extensionsStr.split(",");
                for (int i = 0; i < this.extensions.length; i++) {
                    this.extensions[i] = extensions[i].trim();
                }
            }
            log.debug("Set Policy rule [EXTENSIONS]: "
                    + this.extensions.toString());

        } catch (JaxenException e) {
            throw new RegistryException(
                    "Failed to read Policy resource: " + policyPath, e);
        }
[...]
```

Table 33. Excerpt from ImagePolicy.java

Table 34 contains snippets of code from the implementation of the image policy validator which reads in the properties of the actual image resource being validated at runtime, and compares them against the policy rules. One can notice there is once again a tight coupling between the policy rules and the function of this policy validator. It can only validate a specific set of rules: 1) extensions, 2) max width, 3) max height and 4) max file size. If the structure of the XML policy rules file should change then this code needs to be modified and redeployed.

```
package org.seerc.cast.regrep.validators.app;

[...]
public class ImageValidator extends Validator {

    private static final Log log =
    LogFactory.getLog(ImageValidator.class);
    private ImagePolicy policy;

    public ImageValidator() {
        super();

        try {
            policy = new
ImagePolicy(CastConstants.POLICY_PATH_IMAGES);
        } catch (RegistryException e) {
            log.error("Error initializing Image Policy", e);
        }
    }

[...]
    public ValidationResult validate(Resource resource) {

[...]
        try {

            log.debug("Image file size in bytes: "
                    + resource.getContentStream().available());

            BufferedImage image =
            ImageIO.read(resource.getContentStream());

            log.debug("Validating image width [" + image.getWidth()
                    + "] against  Image Policy maximum width ["
                    + this.policy.getWidth() + "]");
            if (image.getWidth() > this.policy.getWidth()) {
                return this.failed("Image exceeds maximum width of "
                        + this.policy.getWidth());
            }

            log.debug("Validating image height [" + image.getHeight()
                    + "] against  Image Policy maximum height ["
                    + this.policy.getHeight() + "]");
            if (image.getHeight() > this.policy.getHeight()) {
                return this.failed("Image exceeds maximum height of "
                        + this.policy.getHeight());
            }

            log.debug("Checking image extension" + resource.getPath());
```

```
            String extensions[] = policy.getExtensions();
            String pathArr[] = resource.getPath().split("/");
            String filename = pathArr[pathArr.length - 1];
            pathArr = filename.split("\\.");
            String resourceExtension = pathArr[1];
            boolean extensionFound = false;
            for (int i = 0; i < extensions.length; i++) {
                if
(resourceExtension.equalsIgnoreCase(extensions[i])) {
                    extensionFound = true;
                    break;
                }
            }

            if (extensionFound) {
                log.debug("Image extension is valid.");
            } else {
                log.debug("Image extension is invalid.");
                return this.failed("Image extension is invalid.");
            }

            return this.succeed();

        } catch (IOException e) {
            // TODO Auto-generated catch block
            log.debug(e.getMessage());
        } catch (RegistryException e) {
            // TODO Auto-generated catch block
            log.debug(e.getMessage());
        }

        return this.failed("Image validation failed");
    }

[...]
```

Table 34. Excerpt from ImageValidator.java

There are other cases of artefact validation policies in CAST where the policy definition has been implemented in a simpler fashion. In the case of the service interface policy mentioned in section 8.3.3 there is no declarative XML representation of the policy rules and no object converter logic. The policy definition is embedded directly in the implementation of the validation logic. At runtime, a policy validator object reads in the service interface resource (the WSDL file) and applies a check on the number of distinct endpoints as illustrated in the snippet of Table 35 below. Should the "primary/failover server" policy change in the future, the validator logic will need to be revised and redeployed to the platform.

```
package org.seerc.cast.regrep.validators.service;

[...]

public class ServiceInterfaceValidator extends Validator {

    private static final Log log = LogFactory
```

```
    .getLog(ServiceInterfaceValidator.class);

    @Override
    public boolean applyValidation(Resource resource) {

[...]
            // check for two distinct end-points
            for(int i=0;i<wServices.length;i++)
            {
                Set<String> disctinctEndpoints = new HashSet<String>();
                for(Endpoint e:wServices[i].getEndpoints())
                {
                    disctinctEndpoints.add(
                     e.getAddress().toString());
                }
                if(disctinctEndpoints.size() != 2)
                    return this.failed("Each defined service must have
                    two non-identical endpoints.);
            }
[...]
```

Table 35. Excerpt from ServiceInterfaceValidator.java

8.4.2.2 Definition of lifecycle management policies

Artefact validation represents one of the two principal governance functions which are delivered by the CAST platform governance support system. The other function is lifecycle management. In this section we will look at an example of policy definition that allows the platform operator to control the lifecycle of CAST platform solutions.

Section 8.3.3 presents a policy applied to the collection of artefacts and metadata that make up a CAST solution. Before the solution can be allowed to advance from one lifecycle stage to the next, the platform operator needs to validate that the solution collection meets certain criteria and contains a minimum set of valid resources in the form of artefacts and metadata. The constraints applied depend on the lifecycle state that the solution is transitioning to (Development to Testing, Testing to Review, Review to Beta, etc.).

Let us consider a solution transitioning from the state of Testing to the state of Review. Before allowing this to take place, the platform operator needs to ensure that:

- There exists a non-empty text description of the solution
- There exists a valid pricing specification file
- There exists a valid license file
- There exists a valid provider details file
- All apps or services this solution depends on are in beta or production state

In this case, policy definition is a three-step process:

1. Creating/modifying a lifecycle management policy rules file in XML format.
2. Implementing/modifying the lifecycle management logic in Java.
3. Packaging the components as JAR archives, deploying them to the runtime environment, and restarting the server (Java servlet container) to complete the deployment.

Table 36 below shows a snippet from the XML file (SolutionLifeCycle.xml) that allows configuration of the CAST governance support system front-end. The purpose served by this configuration file is only to adapt the governance support system's user interface at run-time, presenting the user with the checks (preconditions) relevant to each lifecycle state of the solution. It plays no part in the further definition of those checks or their actual enforcement, except for also specifying which server-side function should be triggered to carry out the actual policy evaluation.

```
<aspect name="CAST Solution Lifecycle"
class="org.seerc.cast.regrep.lifecycle.SolutionLCM">
    <configuration type="literal">
        <lifecycle>

[...]
    <state name="testing" location="/%organization/solutions/testing">
        <checkitem>Solution description (metadata)</checkitem>
        <checkitem>Pricing specification (pricing.xml)</checkitem>
        <checkitem>License terms (license.txt)</checkitem>
        <checkitem>Provider details (provider.xml)</checkitem>
        <permissions>
            <permission action="demote" roles="developer" />
            <permission action="promote" roles="developer" />
        </permissions>
        <js>
            <console promoteFunction="doPromote"
                     demoteFunction="doDemote">
                <script type="text/javascript">
                    doDemote = function() {
                    window.location =
                    "../resource.jsp?path=/solutions/development";}
                    doPromote = function() {
                    window.location =
                    "../resource.jsp?path=/solutions/review"; }
                </script>
            </console>
            <server promoteFunction="doPromote"
                    demoteFunction="doDemote">
                <script type="text/javascript">
                    function doDemote() {
                    return "Solution demoted to development state"; }
                    function doPromote() {
                    return "Solution promoted to review state"; }
                </script>
            </server>
        </js>
    </state>

[...]
        </lifecycle>
```

```
    </configuration>
</aspect>
```

Table 36. Excerpt from SolutionLifeCycle.xml

Table 55 in the appendix provides snippets from the actual lifecycle management logic implemented in Java. As with the case of service interface validation presented in the previous section, policy definition is once again embedded in the concrete implementation of the policy evaluation logic.

To evaluate if a solution is promotable to the next lifecycle state, the lifecycle policy validator will check the conditions applicable to the attempted transition (in this example, moving from Testing to Review). As a first step, it will re-run the checks associated with the transition that came before that (Development to Testing), followed by the checks applicable to the transition at hand (Testing to Review conditions).

The policy is defined by way of implementing checks on these conditions. In this example, the lifecycle policy validator will check to make sure that the description of the solution is not empty and that the solution collection includes a valid pricing specification file, a valid license file, and a valid provider details file. Should all these conditions be satisfied the lifecycle management logic will then check the maturity of the solution's dependencies, i.e. whether all apps and services that the solution comprises are in a normal operational state (beta or production).

As it should be evident, the CAST governance support system does not facilitate defining lifecycle management policies in any way that can be distinguished from how these policies are actually enforced. As a consequence, every time there is demand for a change to how lifecycle is managed there is a need to reengineer the policy evaluation component presented above, test the new code, package it in a JAR archive and have it redeployed to the platform on a scheduled maintenance/downtime window.

8.4.3 Data extraction

In section 8.4.2 above we looked at how policy definition is made possible with the CAST platform registry & repository system. Enforcing artefact validation policies and lifecycle management policies involves extracting a set of data relevant to the resource being governed and making this data available to the policy evaluation logic.

In this section we are looking at how this was implemented with the CAST platform registry & repository system based on the infrastructure provided by the open-source WSO2 Governance Registry platform.

8.4.3.1 Retrieval of data for artefact validation

Unlike other ecosystem scenarios where policy evaluation can be carried out by different ecosystem partners in parallel, the CAST project consortium focused on studying and facilitating an ecosystem scenario where policy evaluation is performed centrally by the CAST platform. The policy evaluator role is not assumed by any other actor except for the platform operator.

The providers of governed resources, such as the ISVs who create CAST platform apps, are the ones who create the data against which policies are evaluated. This data is typically specification files or binary files like the examples presented in previous sections which are created and uploaded onto the CAST Registry & Repository (R&R) system by the resource providers. The CAST R&R system serves as a central repository for cataloguing and managing those artefacts.

The CAST R&R system also offers a visual interface through which a user can navigate to these resources and perform relevant operations (Figure 18).

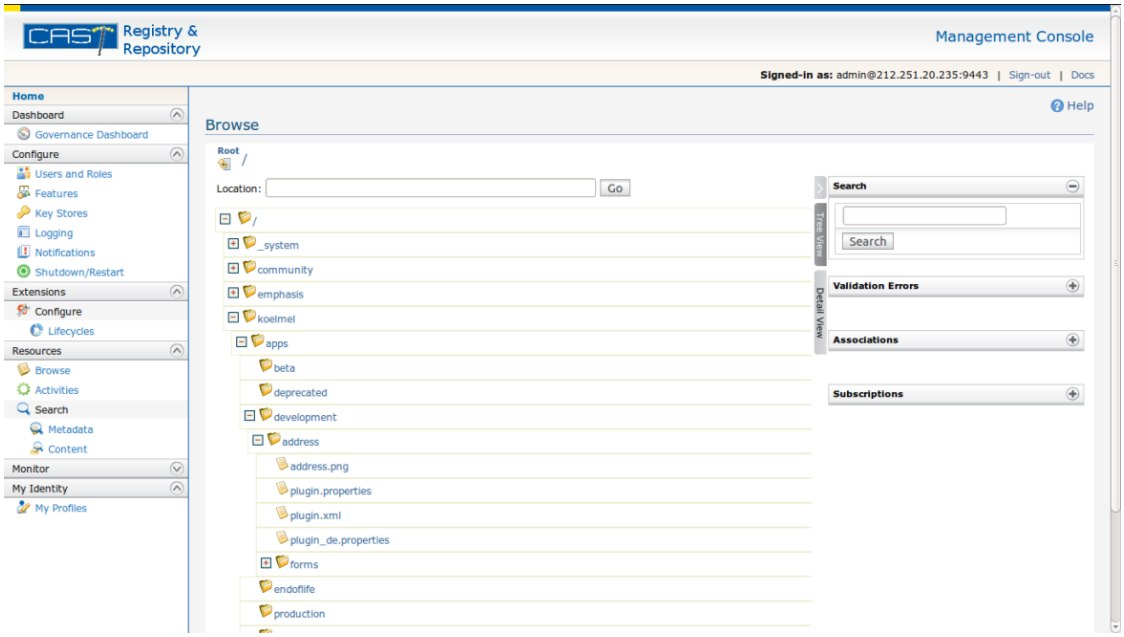

Figure 18. Artefact organisation in CAST Registry & Repository system

At policy evaluation execution time, the R&R system is queried by the policy evaluation logic (i.e. by the implementation of the policy validator component) to retrieve and process those artefacts.

Table 37 provides a snippet from an example artefact we have already seen in previous chapters; a service interface description as a WSDL document. The code executed as shown earlier in Table 35 would directly retrieve the WSDL document from the repository in order to process it.

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns="http://www.w3.org/ns/wsdl"
    targetNamespace=
"http://cast-project.eu/governance/examples/wsdl"

[...]

  <service name="translationService"
       interface="tns:translationServiceInterface">

     <endpoint name="endpoint1"
               binding="tns:translationSOAPBinding"
               address ="http://144.76.8.88"/>

     <endpoint name="endpoint2"
               binding="tns:translationSOAPBinding"
               address ="http://143.167.8.2"/>
  </service>

</description>
```

Table 37. Excerpt from service interface description artefact (WSDL)

Table 38 illustrates one more example of an artefact under governance. It is a pricing definition file for a solution deployed on the platform (as described in section 8.3.3.1).

```
<?xml version="1.0" encoding="UTF-8"?>
<price xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://localhost/schema/pricing.xsd"
xmlns="http://localhost/schema">
    <amount>35</amount>
    <modality>per user, per month</modality>
</price>
```

Table 38. Pricing definition artefact (XML)

In addition to those cases where the artefacts under governance are represented as files residing in the repository, validation policies could also refer to metadata that is stored in the relational database backend of the CAST Registry & Repository system. For instance, checking that the description of a solution is not empty (as in section 8.3.3.1) involves querying the system's database. This is done through the data access APIs provided by the WSO2 Governance Registry platform, which are invoked from within the Java-based implementation of the policy validator object.

In either case (artefacts as files, or artefacts as metadata) the CAST Registry & Repository system does not offer any intermediate layer to provide any abstraction over how these artefacts are retrieved. As a consequence, should the structure of any of these artefacts ever change, the artefact validation logic implemented by all relevant policy validators would be critically affected and would also need to change. This would

require modifying all the policy validators which are affected by this change and redeploying them to the CAST platform.

8.4.3.2 Retrieval of data for lifecycle management

The way in which data is retrieved for the purposes of lifecycle management is not much different to how this is performed for the purposes of artefact validation.

In some cases, transitioning to a new lifecycle state may involve retrieving artefacts from the repository and running them through checks. In other cases, it may only involve retrieving and checking data from the CAST Registry & Repository system backend. In either case, the data retrieval function is embedded in the lifecycle management components. There is no abstraction layer that allows the data representation to evolve independently from the policy or the policy evaluation logic. Changes to how data is represented will trigger a need to reengineer the policy validators.

8.4.4 Policy evaluation

In the previous two sections we looked at how the CAST platform registry & repository system supports the definition of policies and the extraction of the data relevant to each resource or lifecycle being governed. We also illustrated how this data is evaluated against the relevant policies.

We highlighted the tight coupling between those two functions (defining policies and extracting data for evaluation) which are effectively embedded inside the implementation of the policy evaluation logic. In this section we explain how this policy evaluation logic is invoked and the development and configuration activities that are required to enable this. We also discuss the implications of this process in terms of software modifiability.

For completeness, we also provide a closer look at how the outcome of policy evaluation is visually communicated to the user of the CAST platform registry & repository system.

8.4.4.1 Evaluation of artefact validation policies

Once artefact policy validators are implemented in the way discussed in section 8.4.2, and packaged as JAR archives to be deployed to the CAST Registry & Repository runtime, they can be invoked by the system at the time they are needed**.** This invocation is based on triggers which are either system events or user actions.

For instance, invocation of a policy validator can happen during cataloguing of new CAST platform entities and storage of associated artefacts, when saving, modifying or

deleting a resource, during the creation, modification or deletion of dependencies among entities and during lifecycle management, when promoting the state of a managed resource.

The way this is supported by the WSO2 Governance Registry, which provides the infrastructure for the CAST governance support system, is through an extension architecture of Handlers and Filters.

A handler is a pluggable Java component that extends the functionality of the WSO2 Governance Registry platform to allow customisation of how resources are processed. The methods provided by handlers are invoked by the platform based on event triggers such as creating a new resource, updating it or adding it to a collection. Every handler has an associated filter. Filters provide the criteria for engaging handlers. If a filter's criteria evaluate to true, the associated handler will be invoked. For example, we can set a filter that detects actions on resources of a specific media type (MIME type) such as XML files in general, or WSDL files specifically.

In brief, a filter invokes a handler and the handler in turn invokes the respective artefact policy validator. For instance, uploading a new artefact like the one shown in Table 37 whose MIME type is `application/wsdl+xml` will trigger invocation of the handler for WSDL files, which will in turn invoke all relevant WSDL-related policy validators, such as the "double-endpoint" policy validator shown in Table 35.

Before handler and filter components can be used to trigger artefact policy validators, they also need to be packaged as JAR archives and deployed to the CAST Registry & Repository system. They also need to be registered with the system by configuration (editing the R&R Registry.xml file).

Figure 19 shows how the CAST platform registry & repository helps visualise errors detected during artefact validation. In the specific instance there are three different artefacts (plugin.xml, sla.xml and solution.xml) which fail validation against their relevant policies.

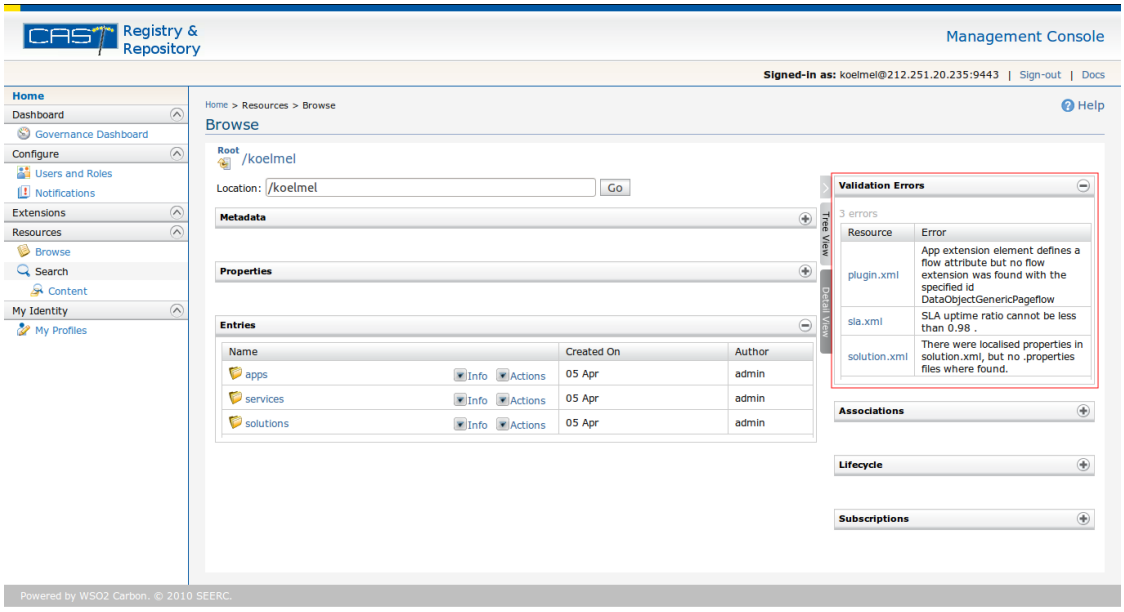

Figure 19. Artefact validation interface

8.4.4.2 Evaluation of lifecycle management policies

Lifecycle modelling offers a bird's eye view of the status of a solution, app or service which is useful to both the platform operator and ecosystem partners, and allows to associate lifecycle transitions with artefact validation policy checks, which are triggered when a user attempts to set some CAST platform software unit to a new state.

The way this is supported by the WSO2 Governance Registry is through an extension architecture of Aspects, which is similar to the architecture of Handlers and Filters discussed above. Aspects can be used to associate custom behaviours with resources. The difference between aspects and handlers is that handlers are automatically applied to a resource, whereas aspects need to be invoked manually following a user's action.

Lifecycle management policy validators are implemented as WSO2 Registry aspects. Before they can be used they need to be packaged as JAR archives and deployed to the CAST Registry & Repository system. They also need to be registered with the system by configuration (editing the Registry & Repository system's Registry.xml file).

The screenshot in Figure 20 indicates that ServiceX is in the state of Testing, with all the required resources for this state checked, and is ready to be promoted to the next state (Review).

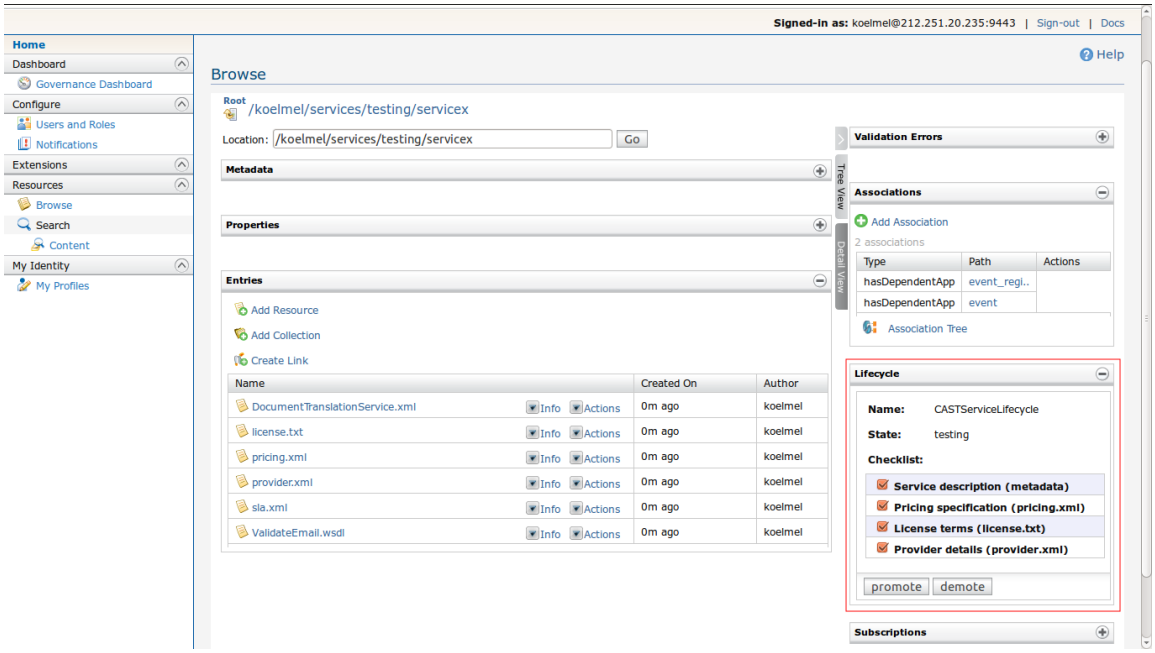

Figure 20. Lifecycle management interface

8.4.5 Remarks

The complete set of development and configuration steps required in order to facilitate evaluation of artefact validation and lifecycle management policies is as follows:

1. Implement the lifecycle policy validator component (as in the example shown in Table 55 of the appendix). Package it as JAR archive.
2. Implement the artefact policy validator component (as in the example of Table 35). In some cases this may involve first creating a policy rule file (as in the example of Table 32) and a Java object converter (as in Table 33). Package it as JAR archive.
3. Implement a new handler component or modify the existing one so that it invokes the new artefact policy validator component. Package it as JAR archive.
4. Implement a new filter component (only needed in case the artefact type is new) which invokes the artefact handler component. Package it as JAR archive.
5. Deploy all JAR archives to the CAST Registry & Repository system runtime.
6. Register the new aspect component and the new filter and handler components with the system by editing its main configuration file (Registry.xml).
7. Restart the CAST Registry & Repository system for the changes to take effect.

It becomes apparent that the above process demands coding against low-level APIs and therefore requires good knowledge of the CAST Registry & Repository system architecture. Such knowledge may be reasonable to assume for the software engineers

working for the CAST platform operator, but this is not the case for third-parties in the ecosystem who, as discussed in 8.3.2, may also wish to act as policy providers.

It is difficult to imagine a third-party ISV implementing handler, filter and aspect (lifecycle management) components against the low level APIs available by the system. Artefact policy validators are implemented against a somewhat higher-level API, so they would in fact represent less of a challenge to construct for third-party ecosystem partners. The rest of the components however seem to be only suited to a development team working for the CAST platform operator.

Not only is this process impractical for third-party ecosystem partners, but it is also challenging from a software reliability point of view for the CAST platform operator and requires a heavy QA and deployment cycle on the operator's end. It also raises concerns from a security perspective, as the CAST platform operator needs to expose information on the internal workings of the CAST Registry & Repository system and its APIs to a great level of detail.

8.5 Description of alternative solution based on PROBE framework

In the scope of the CAST project, a set of 37 different governance policies relating to lifecycle management and artefact validation were captured and analysed. Subsequently, the policy evaluation components to enforce these policies were implemented based on the process described in the previous section. Through this exercise we gained valuable insights into the limitations of state of the art approaches to policy-based governance as implemented by contemporary governance support systems.

In this section we demonstrate an alternative approach to policy-based governance for the CAST platform, based on the new framework that this thesis puts forward – a framework for governance support system development that builds on Semantic Web technologies and Linked Data principles. We show that the new design approach afforded by the PROBE framework can meet the full set of requirements for policy-governance in a complex and dynamic ecosystem environment. We also set the basis to demonstrate its several advantages over the earlier design approach adopted by the CAST project.

8.5.1 Overview

This thesis puts forward a new approach to the definition and enforcement of governance policies, where ontology-based knowledge representation and reasoning is central. The PROBE framework leverages ontologies as both design-time artefacts for policy definition, as well as run-time artefacts for policy evaluation/enforcement.

The foundation for policy definition is a CAST platform governance ontology, which defines common modelling constructs corresponding to the different types of logical entities found on the CAST platform. This includes the different kinds of software units (solutions, apps and services), different kinds of software artefacts (e.g. deployment descriptors, interface definitions, pricing specifications, localisation files, images), artefact collections, lifecycle states (development, testing, review, beta, production, deprecation, end-of-life), and other relevant concepts.

Ecosystem partners who act as policy providers, including the CAST platform operator, import this "global" ontology and extend it to create their own local policy ontologies. The policies defined by each ecosystem partner are saved into that local ontology. The ontology resource is accessible over the web by whoever other ecosystem partner is authorised to read and process it, for policy evaluation or other purposes.

Ecosystem partners who act as data providers create and share descriptions of their governed resources using the terms defined in the same "global" ontology that policy providers also use. This allows policies and governed resources to be defined on the basis of a common vocabulary and facilitates automation for the process of policy evaluation.

The approach we describe is not intended to displace the governance support system infrastructure which is used in the CAST Registry & Repository system. Instead, our design intention is for the PROBE framework to constitute an additional architecture layer on top of systems such as the open-source WSO2 Governance Registry platform – which was used in developing the CAST R&R system.

A concrete description of how the PROBE framework components can be integrated into WSO2 Governance Registry or similar platforms is beyond the scope of this work. Suffice to say that the extension points provided by the architecture of WSO2 Governance Registry in the form of Handlers and Aspects can provide the necessary integration hooks.

8.5.2 Policy definition

In the following subsections we will look at how our new framework allows describing CAST governance policies to facilitate resource governance through artefact validation and process governance through lifecycle management.

8.5.2.1 Definition of artefact validation policies

Section 8.4.2.1 presented how artefact validation policies were defined in CAST. One of the examples used in that section is the policy which states that screenshots of CAST apps should be submitted in either JPG or PNG format, that they should not exceed

1MB in size and that their horizontal and vertical dimensions should be within specific limits.

An alternative way to define that same policy using the PROBE framework is provided in Table 39.

```
Class: ValidAppScreenshot

    EquivalentTo:
        (hasContentType some ({_image/jpeg, _image/png}))
        and (hasSizeInKB some xsd:integer[<= 1024])
        and (hasHeightInPixels some xsd:integer[>=300, <=600])
        and (hasWidthInPixels some xsd:integer[>=400, <=800])

    SubClassOf:
        AppScreenshot,
        hasContentType only ({_image/jpeg, _image/png}),
        ValidAppArtefact
```

Table 39. Definition of positive-form policy ValidAppScreenshot (defined class)

The specific policy presented above is created by the CAST platform operator with the goal of achieving some uniformity in how CAST apps are presented inside the CAST platform app store. We can imagine similar policies being created by other partners in the ecosystem, such as ISVs, to govern other quality attributes of CAST apps or external services being used as building blocks in their own CAST-powered solutions and apps.

Ontology-based artefact validation policies like the one presented above can be authored using an open-source ontology editor tool such as Protégé[51]. Alternatively, they could also be authored through a special-purpose editor which would hide the expressive power and complexity of OWL but leverage the ontology's structure to present the user with a limited set of policy constructor options to choose from, and the respective value input fields.

The artefact validation policy is succinctly expressed in the policy ontology and unlike the original approach described earlier where bespoke processing with Java was required before a policy can be evaluated, this approach will use generic, standard-based OWL processing tools. The ontology-based approach to policy definition allows link maintenance between policies, as well as between policies and governed resource types (governance subjects) which enables traceability. It is an analysis-friendly representation of policies that brings verifiability benefits. The domain-level abstractions used in the encoding of policies aids comprehensibility and the platform-independent policy representation format enables interoperability between ecosystem partners.

[51] http://protege.stanford.edu/

8.5.2.2 Definition of lifecycle management policies

Section 8.4.2.2 presented an example of a lifecycle management policy to govern the transition of a CAST solution from Review to Testing. The policy is created by the platform operator and states the following conditions:

- There exists a non-empty text description of the solution
- There exists a valid pricing specification file
- There exists a valid license file
- There exists a valid provider details file
- All apps or services this solution depends on are in beta or production state

An alternative way to define that same policy using the PROBE framework is provided in Table 40 below.

```
Class: SolutionPromotableToReview

    EquivalentTo:
        SolutionInTesting
        and (hasDescriptionMetadata exactly 1 ValidDescription)
        and (hasCollection exactly 1
               (CollectionOfValidSolutionArtefacts
                and SolutionArtefactsForTransitionToReview))
        and (hasDependency some
                  (PlatformEntityInBeta or
                   PlatformEntityInProduction))

    SubClassOf:
        SolutionInTesting,
        hasDescriptionMetadata only ValidDescription,
        hasDependency only
            (PlatformEntityInBeta or
             PlatformEntityInProduction),
        hasCollection only
            (SolutionArtefactsForTransitionToReview
             and CollectionOfValidSolutionArtefacts)
```

Table 40. Definition of positive-form policy for the transition of a CAST solution to the review stage

The `EquivalentTo` expression states that for a solution to be promotable to Review stage it must currently be in Testing stage (`SolutionInTesting`); must have a unique valid description (`ValidDescription`, as per Figure 11); must have a unique collection of valid artefacts (`CollectionOfValidSolutionArtefacts`) including all artefacts necessary for a transition to Review (`SolutionArtefactsForTransitionToReview`); any other software units the solution depends on must be in a stage which is fully operational and in a live environment (`PlatformEntityInBeta` or `PlatformEntityInProduction`).

The policy is defined in a compositional fashion, making references to other separately defined classes:

```
ValidDescription,
CollectionOfValidSolutionArtefacts,
SolutionArtefactsForTransitionToReview,
PlatformEntityInBeta,
PlatformEntityInProduction
```

The definition of `ValidDescription` is not given directly, but indirectly through the negative-form policy for `InvalidDescription` already presented in Table 11 of section 5.3.2.2.

Table 41 provides the description of the `CollectionOfValidSolutionArtefacts` policy module.

```
Class: CollectionOfValidSolutionArtefacts

    EquivalentTo:
        (contains some ValidSolutionArtefact)
         and (contains only ValidSolutionArtefact)

    SubClassOf:
        SolutionCollection
```

Table 41. Definition of CollectionOfValidSolutionArtefacts

The necessary and sufficient conditions of the class state that individuals belonging to this class must be connected along the `contains` property to at least one `ValidSolutionArtefact`, and that all connections to anything along the `contains` property should be exclusively to individuals known to be `ValidSolutionArtefact`. In other words, a collection should contain at least one solution artefact and nothing else but valid solution artefacts. The definition of `ValidSolutionArtefact` is given indirectly, defined as superclass of `ValidSolutionLicence`, `ValidSolutionPricing`, `ValidSolutionProviderDetails` and other relevant solution artefact policy definitions.

Table 43 provides the description of `SolutionArtefactsForTransitionToReview`.

```
Class: SolutionArtefactsForTransitionToReview

    EquivalentTo:
        SolutionArtefactsForTransitionToTesting

    SubClassOf:
        SolutionCollection
```

Table 42. Definition of SolutionArtefactsForTransitionToReview

The policy module provided by this definition states that the solution artefacts which are necessary for a transition to review are exactly the same as those necessary for a transition to testing. The definition of `SolutionArtefactsForTransitionToTesting` is provided in Table 43.

```
Class: SolutionArtefactsForTransitionToTesting

    EquivalentTo:
         (contains some ValidSolutionLicence)
         and (contains some ValidSolutionPricing)
         and (contains some ValidSolutionProviderDetails)
    SubClassOf:
        (contains exactly 1 ValidSolutionLicence)
         and (contains exactly 1 ValidSolutionPricing)
         and (contains exactly 1 ValidSolutionProviderDetails)
```

Table 43. Definition of SolutionArtefactsForTransitionToTesting

The definition above provides the last building block to define the policy module for `SolutionArtefactsForTransitionToReview`. It states that the solution artefacts which are necessary for a transition to testing comprise a valid solution licence file, a valid pricing spec, and valid administrative contact details for the provider.

```
Class: PlatformEntityInProduction

    EquivalentTo:
        hasLifecycleStateClassification value _Production

    SubClassOf:
        PlatformEntityInState
```

Table 44. Definition of PlatformEntityInProduction

Finally, Table 44 above provides the last policy module definition to completely unfold the policy for `SolutionPromotableToReview`. It defines `PlatformEntityInProduction` by making use of a simple object property (`hasLifecycleStateClassification`) and a constant value for that property.

These examples show how lifecycle management policies can be described in a declarative way, rather than encoded implicitly in the procedural abstractions of the Java routines that validated the same policies in the CAST approach. Furthermore, the ontology language allows classification of similar kinds of lifecycle management, and definition of new policies by extension, which was not possible in the original approach. Compositionality in policy definitions allows information hiding at different levels of abstraction and reusability of policy specification fragments, which also helps with change localisation and policy maintenance.

8.5.3 Data extraction

For policy evaluation to be feasible through a generic and universal method, the heterogeneous platform resources that are subject to governance are described in an abstract and homogeneous manner. Descriptions are extracted from the multiple forms in which platform resources are natively represented to create Linked Data, using the CAST platform governance ontology as the main reference vocabulary.

Section 6.3 presents the method for creating and sharing descriptions of governed resources based on a common ontology. Let us provide an example of applying this method. We will use an artefact related to one of the validation policies we have already seen earlier in this chapter: the app "visual assets" policy from section 8.5.2.1. To validate an app screenshot image file against that policy, one first needs to extract the relevant metadata from the binary image file and make them available in a suitable RDF-based representation.

This task can be performed by whichever ecosystem partner is the original resource provider, i.e. the creator of the app. Alternatively it can be performed by the CAST platform operator who is currently managing the CAST app, and also has access to a copy of the app screenshot artefact inside the CAST R&R system. Once a transformation mechanism is available to extract the important image metadata from the binary file, the mechanism can be reused by different partners in the ecosystem.

Let us assume the app screenshot in question is a JPG image file named `screenshot-732.jpg` with size of 512 KB and dimensions of 300x400 pixels. The resulting output RDF representation of the artefact (after conversion) is shown in Table 45 below.

```
< http://kourtesis.net/phd/2016/examples#screenshot-732 >
< http://www.w3.org/1999/02/22-rdf-syntax-ns#type >
< http://ecosystem-governance.com/ontology#AppScreenshot > .

< http://kourtesis.net/phd/2016/examples#screenshot-732 >
< http://ecosystem-governance.com/ontology#hasSizeInKB >
"512"^^http://www.w3.org/2001/XMLSchema#integer .

< http://kourtesis.net/phd/2016/examples#screenshot-732 >
< http://ecosystem-governance.com/ontology#hasHeightInPixels >
"300"^^http://www.w3.org/2001/XMLSchema#integer .

< http://kourtesis.net/phd/2016/examples#screenshot-732 >
< http://ecosystem-governance.com/ontology#hasWidthInPixels >
"400"^^http://www.w3.org/2001/XMLSchema#integer .

< http://kourtesis.net/phd/2016/examples#screenshot-732 >
< http://ecosystem-governance.com/ontology#hasContentType >
< http://ecosystem-governance.com/ontology#_image/jpeg > .
```

Table 45. Description of app screenshot resource (screenshot-732.jpg) metadata in RDF triples

The RDF triples presented above could be serialised in static RDF/XML files and placed on a web server. Ideally though, they should be persisted in an RDF triple store which exposes a SPARQL query interface. In the latter case, any (ecosystem-authorised) governance policy evaluation engine could query the RDF triple store and retrieve the relevant information on demand.

Table 46 presents an example SPARQL query to obtain all information which is part of the RDF graph at the governance data provider's end, and relates to the app screenshot description.

```
SELECT DISTINCT ?predicate ?object
WHERE
{
   <http://kourtesis.net/phd/2016/examples#screenshot-732>
   ?predicate
   ?object
}
```

Table 46. SPARQL query to retrieve description of app screenshot resource (screenshot-732.jpg)

The SPAQRL query above will return the following description of Table 47 (shown here in Turtle syntax).

```
@prefix : < http://kourtesis.net/phd/2016/examples#> .
@prefix gov: <http://ecosystem-governance.com/ontology#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@base <http://kourtesis.net/phd/2016/examples> .

:screenshot-732  rdf:type  gov:AppScreenshot ;
:screenshot-732  gov:hasSizeInKB  "512"^^xsd:integer ;
:screenshot-732  gov:hasHeightInPixels  "300"^^xsd:integer ;
:screenshot-732  gov:hasWidthInPixels  "400"^^xsd:integer ;
:screenshot-732  gov:hasContentType  gov:_image/jpeg .
```

Table 47. Example description of app screenshot resource (screenshot-732.jpg)

While this data extraction and conversion to RDF format may sometimes form an additional stage in the PROBE approach, the advantage of this is that it only needs to be converted once, by the responsible data provider, after which the declarative representation of the same data will be available to all ecosystem partners. Furthermore, conversion mechanisms, once established for each raw data source, may be reused to convert all data of the same type.

8.5.4 Policy evaluation

The PROBE framework method to evaluate governance policies based on DL reasoning is discussed in detail in section 7.3.

The process is initiated when the policy evaluation engine is given a URI pointing to a description of the governed resource. In the app screenshot example in Table 47 above the URI is `http://kourtesis.net/phd/2016/examples#screenshot-732.`

Once the policy evaluation engine obtains the resource description shown in Table 47 it determines the applicable policies by looking at the resource's `rdf:type` axiom. The specific artefact is of type `gov:AppScreenshot` which suggests the relevant policies are the following subclasses of the `AppScreenshot` policy class:

```
http://ecosystem-governance.com/ontology#ValidAppScreenshot
http://ecosystem-governance.com/ontology#InvalidAppScreenshot
```

To check if the RDF data from Table 47 satisfies the definition of `ValidAppScreenshot` shown in Table 39 (or, by extension, satisfies the mutually disjoint definition of `InvalidAppScreenshot),` the PROBE policy evaluation engine will invoke the instance checking function of an OWL DL reasoner such as Pellet [127] or Hermit [128].

As a pre-processing step, before invoking the reasoner, the policy evaluation engine will import the RDF description and construct an OWL individual to represent the described governed resource. It will then run a generic closure axiom generation algorithm as detailed in section 7.3 to automatically add additional local closure axioms along the following salient properties of the resource being described:

```
http://ecosystem-governance.com/ontology#hasSizeInKB
http://ecosystem-governance.com/ontology#hasHeightInPixels
http://ecosystem-governance.com/ontology#hasWidthInPixels
http://ecosystem-governance.com/ontology#hasContentType
```

The algorithm examines the equivalence class axiom representing the policy of interest, determines which (asserted or inferred) properties are relevant for classification, constructs anonymous type assertions with the exact known objects or literals per each property of importance, and adds those to the object to be checked. The generic instance checking method of the DL reasoner is invoked and results returned to the system that queried the policy evaluation engine.

Policy checking in PROBE is therefore performed by industry-standard OWL-DL reasoners that work directly with the chosen description logic. This is a considerable improvement over the bespoke Java algorithms that were used to check idiosyncratic representations of the data in the CAST approach, since checking policies is fully independent of policy creation, and may be carried out by different partners in the ecosystem.

8.5.5 Remarks

In the interest of making a comparison of the two design approaches easier for the reader, this section used the same policy examples as the ones illustrated in section 8.4.

Specifically, we used the examples of (1) validating an app screenshot artefact; and (2) managing the lifecycle transition of a CAST solution from testing to review stage.

These policies represent two examples from the set of 37 governance policies analysed in the scope of project CAST [3],[22]. Beyond what is presented in this section, the PROBE framework has been successfully applied to defining and evaluating the complete set of those policies.

One of the questions that surfaced on the outset of this work was whether the expressivity of OWL2 DL (i.e. $\mathcal{SROIQ}^{(\mathcal{D})}$) would prove sufficient for representing all of the policies in the CAST project dataset as OWL class axioms, or whether it would be necessary to step outside OWL2 DL boundaries. Indeed, the representation of some CAST policies proved to be demanding in terms of expressivity, and required the use of some SWRL rules. For most of the policies the less expressive DL $\mathcal{ALCOIQ}^{(\mathcal{D})}$ has been sufficient. The need to step outside DL and include Horn-clauses encoded in SWRL was mitigated by the easy integration of SWRL with OWL. SWRL rules and OWL ontologies share a common semantics, and can be serialised together. Furthermore, SWRL is a de facto standard supported by many DL reasoning engines such as Pellet [127] or Hermit [128]. Therefore, combining OWL DL axioms and SWRL rules does not present any theoretical or practical obstacles from the perspective of policy evaluation.

8.6 Comparative assessment of design approaches

In this chapter we have so far presented the application of two different software design approaches on a common case study: developing a governance support system for CAST-powered cloud service ecosystems.

In this section we evaluate and compare the two design approaches based on the requirements set forth in section 3.4.

In an analysis of case study research methodology, Yin [179] defines comparative case studies as methods where "the same case is repeated twice or more to compare alternative descriptions, explanations, or points of view". In the context of software engineering research the alternative descriptions in a comparative case study represent alternative design approaches or techniques applied on the case [180] so as to evaluate them in terms of suitability for a given purpose. This evaluation includes highlighting not only differences between the approaches but also trade-offs, and is based on common, predefined "units of analysis" [181].

8.6.1 Units of analysis

We examine the two design approaches from the perspective of the different roles involved in the ecosystem governance process:

1. Policy provider role
2. Data provider role
3. Policy evaluator role

The analysis is grouped based on the different types of concerns associated with each role, relating to manageability and evolvability of their individual functions. Based on the requirements analysis in section 3.4, there concerns are classified as follows:

Policy provider role concerns:

- Ease of creating, testing and maintaining policies
- Ease of managing knowledge about policies
- Limiting the impact of policy provider changes to other roles
- Limiting the impact of external changes to the policy provider role

Data provider role concerns:

- Ease of creating, maintaining and sharing descriptions of governed resources
- Ease of managing knowledge about resources and resource descriptions
- Limiting the impact of data provider changes to other roles
- Limiting the impact of external changes to the data provider role

Policy evaluator role concerns:

- Ease of evaluating conformance of resource descriptions to policies
- Ease of managing knowledge about relationships between policies and resources
- Limiting the impact of policy evaluator changes to other roles
- Limiting the impact of external changes to the policy evaluator role

8.6.2 Scenario-based comparison

To evaluate the two design approaches we adopt a scenario-based comparison methodology inspired by Bengtsson's method for architecture-level modifiability analysis (ALMA) [182]. ALMA is a method to analyse the modifiability potential of a software system based on the characteristics of its architecture. The method is scenario-based and is used by software engineers and business information system

analysts to compare software architecture candidates, assess the risk associated with modifications of the architecture, or predict the effort needed to implement anticipated modifications [182].

The scenarios employed for this kind of analysis are "change-scenarios", which capture future events that will require a system to be adapted. In the context of our comparison, change scenarios help to evaluate the two design approaches in terms of how they support both evolvability and manageability of the governance process.

To elicit some interesting change scenarios, it is useful to consider potential changes from the perspective of the three roles in the governance process and their associated process elements. Generally, the three elements of a governance process that may evolve over time are governance policies, governed resources, and policy evaluation engines. In the following change scenarios we consider some interesting cases of changes to those process elements, without exhausting all possible combinations.

Before we proceed with describing scenarios, we introduce the following example ecosystem partners to set a common context across the scenarios:

- *Partner A.* Provider of web APIs (external web services) which are used by other ecosystem partners to create CAST apps. As the provider of such ecosystem resources Partner A is also a Data Provider, i.e. is responsible to provide descriptions of the resources to other ecosystem partners for the purposes of policy evaluation.
- *Partner B.* Creator of a CAST app which utilises the external service provided by Partner A as well as external services provided by other ecosystem partners. Partner B is also a Policy Provider. Its policies govern different properties of external services, such as interfaces and the encryption standards used to communicate with those services.
- *Platform.* In its capacity as the ecosystem facilitator the CAST platform operator acts as both a Policy Provider and Policy Evaluator.

8.6.3 Change scenarios

We consider the following four indicative change scenarios.

1. A new policy is introduced over an existing type of artefact.
2. A new artefact type is introduced under an existing set of policies.
3. A new policy is introduced over a new type of artefact.
4. A new policy evaluation engine implementation is introduced.

In the sections that follow we walk through the actions to be taken by ecosystem partners in each scenario, for each of the two different design approaches we are looking to compare.

8.6.3.1 Change scenario 1. A new policy is introduced over an existing type of artefact

In this scenario, Partner B introduces a new policy placing constraints on the external services used by its apps. The policy states that it is mandatory for external services used by its apps to offer REST protocol bindings. The new policy affects the service interface description artefacts (WSDL documents) offered by service providers such as Partner A. Enforcement of the policy is carried out by the CAST Platform operator.

Scenario 1	**Change implementation process**	**Execution-time process**
CAST design approach	Partner B implements a new policy validator component in Java for the new REST protocol binding policy. The new component is tested and packaged as a JAR file which is submitted to the Platform. The Platform modifies the implementation of the WSDL artefact handler so that it triggers the new policy validator created by Partner B. Policy evaluation becomes effective after both JAR files pass QA and are deployed to the CAST R&R runtime at a scheduled maintenance window (downtime).	A user of the governance support system takes an action which triggers validation of a WSDL instance document. The WSDL artefact handler triggers the new policy validator to check for REST protocol bindings in the WSDL file.
PROBE design approach	Partner B encodes the new REST protocol binding policy as an OWL class in its local ontology. No change is required to the Platform. Policy evaluation becomes effective immediately.	A user of the governance support system takes an action which triggers validation of a WSDL instance document. A generic artefact handler identifies the WSDL document as a service interface description. A standard process is triggered to fetch all relevant policies from the policy ontology of Partner B and the RDF representation of the WSDL document from Partner A. Policies and RDF data are processed by a generic policy evaluation engine.

Table 48. Analysis of change scenario 1

8.6.3.2 Change scenario 2. A new artefact type is introduced under an existing set of policies

In this scenario, the CAST Platform operator requires that all external service providers such as Partner A describe the interfaces of the web APIs that they offer using WSDL 2.0 specifications, in addition to their existing support for WSDL 1.1. Partner A needs to comply with this requirement. Moreover, the scope of all relevant policies already created by the Platform operator and other ecosystem partners needs to be extended to cover this new type of service description artefact. For instance, one of the Platform's policies states that the interface specification of every external web service should contain two different endpoint URLs (primary and backup server endpoints). This policy was originally intended for use in connection with WSDL v1.1 specifications and now needs to be extended to include service interface descriptions which are based on the WSDL 2.0 standard. The same applies for WSDL-related policies created by other ecosystem partners such as Partner B which created the policy for REST protocol bindings mentioned in Scenario 1.

Scenario 2	**Change implementation process**	**Execution-time process**
CAST design approach	Partner A makes internal changes to be able to provide new WSDL 2.0 type documents as service interface descriptions. The Platform modifies/extends the existing Java implementation of the policy validator for double endpoints, in order to be able to additionally process and check WSDL 2.0 type documents. It repeats the same type of modification/extension for all other relevant policy validator components. Partner B does the same for the REST bindings policy mentioned in scenario 1. So do all other ecosystem partners who have created WSDL-related policies in the past. When all affected ecosystem partners including Partner B have completed their changes, the Platform implements a new handler for WSDL 2.0 type documents in Java, which will trigger all relevant policy validators submitted by partners such as Partner B, and those of the Platform itself. All components are packaged as JAR files. Policy evaluation becomes effective after all new JAR files pass QA and are deployed to the CAST R&R runtime at a sscheduled maintenance window (downtime).	A user of the governance support system takes an action which triggers validation of a WSDL 2.0 instance document. The WSDL 2.0 handler component triggers the modified policy validators to check the WSDL 2.0 file for double endpoints, REST bindings, etc.

PROBE design approach	Partner A makes the relevant internal changes to be able to produce an RDF representation of WSDL 2.0 type documents. No change to the Platform is necessary, except for registering WSDL 2.0 type artefacts as service interface description type artefacts with the generic artefact handler. This is done to trigger validation of WSDL 2.0 descriptions against any applicable service interface policies. Policy evaluation becomes effective immediately.	A user of the governance support system takes an action which triggers validation of a WSDL 2.0 instance document. A generic artefact handler identifies the WSDL 2.0 document as a service interface description. A standard process is triggered to fetch the RDF representation of the WSDL document from Partner A and any policy relevant to service interfaces from the Platform and Partner B policy ontology. Policies and RDF data are processed by a generic policy evaluation engine.

Table 49. Analysis of change scenario 2

8.6.3.3 Change scenario 3. A new policy is introduced over a new type of artefact

In this scenario, Partner B introduces a new policy to govern the encryption standards for all external services used by its apps. The policy states that every external service used by its apps should support asymmetric message-level encryption. Details of the supported encryption parameters should be provided in a WS-Policy document offered by service providers such as Partner A. The policy is to be enforced by the Platform in the scope of lifecycle management and artefact validation for apps created by Partner B.

Scenario 3	**Change implementation process**	**Execution-time process**
CAST design approach	Partner A makes the relevant internal changes to be able to provide new WS-Policy type documents as service interface descriptions. Partner B implements a new policy validator component in Java to check for message-level encryption in WS-Policy type documents. This is packaged as a JAR file and submitted to the Platform. The Platform implements a new handler component for WS-Policy type documents in Java to trigger the relevant policy validators like the one submitted by Partner B, and any validator of the Platform itself. Policy evaluation becomes effective after all component JARs pass QA and are deployed to the CAST R&R runtime at a scheduled maintenance window.	A user of the governance support system takes an action which triggers validation of a WS-Policy instance document. The new WS-Policy handler component triggers the new message-level encryption policy validator.

PROBE design approach	Partner A makes the relevant internal changes to be able to produce RDF representation of WS-Policy documents. Partner B encodes the new message-level encryption policy as an OWL class in its local ontology. No change to the Platform is necessary, except for registering WS-Policy type artefacts as service interface descriptions artefacts with the generic artefact handler. Policy evaluation becomes effective immediately, as long as both Partner A and Partner B have independently completed their changes.	A user of the governance support system takes an action which triggers validation of a WS-Policy instance document. A generic artefact handler identifies the WS-Policy document as a service interface description. A standard process is triggered to fetch all relevant policies from the policy ontology of Partner B and the RDF representation of the WS-Policy document from Partner A. Policies and RDF data are processed by a generic policy evaluation engine.

Table 50. Analysis of change scenario 3

8.6.3.4 Change scenario 4. A new policy evaluation engine implementation is introduced

In this scenario, the Platform operator re-engineers its policy evaluation infrastructure to improve operational efficiency. Changes do not affect the standards used for data exchange between ecosystem partners (such as WSDL or WS-Policy from the previous scenarios). What gets modified is the implementation of the governance support system. The Platform operator upgrades to a newer version of the Registry & Repository system which offers better performance but brings the challenge of providing new Java APIs for developing policy validators and artefact processors.

Scenario 4	Change implementation process	Execution-time process
CAST design approach	The Platform re-engineers the Java-based implementations of all of its policy validators (such as the double endpoint policy validator from scenario 2) to work with the new Registry & Repository system APIs. Partner B does the same for the REST bindings policy validator from scenario 1 and the message-level encryption policy from scenario 3. Every other policy provider in the ecosystem also re-engineers their policy validators. When all affected ecosystem partners including Partner B have completed and submitted their re-engineered policy validators, the Platform creates/modifies all Java-based artefact handler components	A user of the governance support system takes an action which triggers validation of some artefact. The newly re-engineered handler component for the specific artefact type triggers the new policy validators.

	which trigger policy validators created by ecosystem partners such as Partner B, or created by the Platform itself. All components are packaged as JAR files. Policy evaluation with the new version of the CAST R&R system becomes effective after all new JAR files pass QA and are deployed to the R&R runtime at a scheduled maintenance window.	
PROBE design approach	Partner B or other ecosystem partners who are policy providers do not have to make any changes to their policies. The Platform creates/modifies a generic Java-based artefact handler that supports ontology-based mappings between policies and resources. Policy evaluation becomes effective immediately, as long as the Platform makes the existing resource-to-policy mappings available to the new generic artefact handler.	A user of the governance support system takes an action which triggers validation of some artefact. A generic artefact handler identifies the type of artefact. A standard process is triggered to fetch all relevant policies from the policy ontologies and the RDF representation of the artefact from the relevant partners. Policies and RDF data are processed by a generic policy evaluation engine.

Table 51. Analysis of change scenario 4

8.6.4 Comparison of approaches

In this section the two design approaches are compared based on how well they meet different types of concerns. How well does each design approach serve manageability of the governance process? How easy is it for the process to evolve? How do local changes inside the organisation boundaries of a partner affect other ecosystem partners?

Change scenario	**Type of concern**	**Comparison of design approaches**
Scenario 1	Evolvability	With the current design approach adopted by CAST, introducing a new policy requires changes in the form of new software development by both Partner B and the Platform. Policy evaluation becomes effective following a heavy QA and deployment cycle. With PROBE, changes remain entirely local to Partner B. No change is required to the Platform. Policy evaluation can be initiated immediately and there is no downtime for the CAST platform.
	Manageability	With the current CAST design approach, policies are encoded in Java. This means they are not easily understood by the domain experts who issue these policies. Because of the procedural manner in which policies are defined, it is unlikely to reuse policy definitions in creating new ones, or to achieve any substantial degree of modular policy composition. With PROBE, no programming is involved. Policies are encoded in

		a different way, which is similar to using logic rules. This makes them accessible to experts who are not programmers and also makes them amenable to automatic analysis. It also allows automatic checks for policy self-coherence which helps with logical debugging and minimising errors.
Scenario 2	Evolvability	With the current CAST design approach, introducing a new type of artefact such as WSDL 2.0 requires changes to all policy providers who have created WSDL-related policies in the past. Policy evaluation becomes effective following a heavy QA and deployment cycle. With PROBE, change is mostly localised to Partner A. The Platform only needs to be updated with a new mapping that declares the new WSDL 2.0 artefact type as a service interface description. Policy evaluation becomes effective immediately, without platform downtime.
	Manageability	With the current CAST design approach, the changes required at the platform level to support this scenario are extensive. There is no easy way to manage the associations between an artefact type, such as WSDL 2.0 which is used in this scenario, and the relevant policy validators. There is no easy way to capture the information that WSDL 2.0 represents a service interface description, and to deduce that by virtue of this, it relates to a specific set of policies. This information is hidden in the Java code of the WSDL 2.0 processor component which triggers the policy validators. With PROBE, changes at the platform level are much easier to make as they involve no programming (coding cannot be avoided for Partner A in either of the two solution approaches). In addition, because of the knowledge management capabilities inherent in ontology-based policy representation, it is easy to capture associations between policies and governed resources. This allows quickly answering questions like: "Which are the policies relevant to this type of governed resource description?"
Scenario 3	Evolvability	With the current CAST design approach, introducing a new policy over a new type of artefact requires changes by both the data provider (Partner A), the Policy Provider (Partner B) and the Platform. Policy evaluation becomes effective following a heavy QA and deployment cycle. With PROBE, changes are independently localised to Partner A and Partner B. There is no structural change at the Platform level. Policy evaluation becomes effective immediately without platform downtime.
	Manageability	With the current CAST design approach policies are defined by way of implementing the policy rules directly in Java as checks against an artefact or any other form of relevant governance data. This hinders comprehensibility, reusability and verifiability. With PROBE, changes are much easier to make as they involve no programming (coding cannot be avoided for Partner A in either of the two solution approaches). The advantages mentioned in the analysis of scenario 2 and 3 above are applicable here too.

Scenario 4	Evolvability	With the current CAST design approach, introducing a new policy evaluation engine implementation (such as an upgraded Registry & Repository system) requires re-engineering of the entire set of policy validators created by all relevant ecosystem partners. Policy evaluation becomes effective following the heaviest type of QA and deployment across all the scenarios examined. With PROBE, changes are minimal and remain local to the Platform. There is no change to the core of the policy evaluation process which relies on a generic policy evaluation engine.
	Manageability	With the current CAST design approach, the policy evaluator needs to implement custom policy evaluation components (policy validators), from scratch, for every different pair or resource/policy. With PROBE, the policy evaluation logic is free from couplings to specific governance policies and free from couplings to specific types of governance resource description. This design approach offers a way to maintain policy evaluation logic in a form that is generic and reusable.

Table 52. Comparison of design approaches based on change scenarios

8.6.5 Discussion

The defining questions for the evaluation of the two alternative design approaches are: Does the design approach promote effectiveness and efficiency in the internal management of the governance process by different roles? Does it promote evolvability of the governance process by the individual roles? Does it promote both at the same time?

From the foregoing it becomes clear that the design approach which is put forward by PROBE for developing governance support systems has several important advantages over the one originally adopted in the scope of the CAST project.

In the governance support system developed for CAST based on WSO2 Governance Registry, the place where policies are defined, where relevant data is extracted and where policy evaluation logic is applied coincide in the same architectural element: the policy validator component. This has some critical implications with respect to how easy it is for the CAST Registry & Repository system to support changing governance requirements. It limits the ability of ecosystem partners to evolve and therefore limits the potential of the ecosystem to create value through co-development.

There are certainly trade-offs. The advantages afforded by PROBE as a design approach come with a cost. Firstly, software engineers who can code policy validators in Java are easier for a company to recruit compared to ontology engineers who can encode policies in OWL. Secondly, the additional layer of technology components required to implement a solution based on the PROBE framework introduce extra complexity and represent moving parts in the architecture that require additional effort to manage. Thirdly, the level of indirection introduced to hide resource descriptions in

native formats behind RDF descriptions increases complexity and introduces room for error in transformations.

On the other hand, in very pragmatic terms, it is simply not practical for ISVs in the CAST platform ecosystem to write and package Java code in order to have their policies evaluated; and to have this code go through a QA process before it is deployed in a convenient release cycle to the run-time environment of the CAST platform. It is not practical, neither for the ISVs in the ecosystem or for the CAST platform operator.

Moreover, the type of ecosystem envisaged by the CAST project consortium at the time the project was launched does not represent the most advanced form of cloud service ecosystem. We can envisage scenarios which are more advanced and more complex than those emerging from the CAST platform case study. For instance, policy evaluation in CAST is always performed by a single actor, the CAST platform operator, which also acts as policy provider and data provider. We may however envisage ecosystem scenarios where policy evaluation is also undertaken by some of the ISVs in the network, and other much more complex scenarios in terms of governance role distribution. One such example could be the cloud service brokerage scenario as described in scenario #5 from section 3.2.5.

Based on these facts, we argue that PROBE represents an equally feasible but much more suitable design approach for developing governance support systems compared to what contemporary tools can offer; a design approach that is natively suited to the task of supporting governance in a dynamic cloud service ecosystem.

8.7 Summary

In this chapter we present a comparative case study on governance support system design for cloud service ecosystems. The goal of the case study is to evaluate two alternative design approaches for developing governance support systems. The first design approach is the one that project CAST adopted to develop the governance support system for the CAST cloud application platform. The second design approach is the one adopted by the PROBE framework as introduced in chapters 5 to 7 of this dissertation.

The CAST project was set up to investigate the engineering challenges associated with creating a PaaS platform that enables ecosystem-oriented development of business software solutions [3]. The design of the CAST platform is geared towards creating network effects [176], by fostering the emergence of an ecosystem of business software creators around the PaaS [177]. To promote this objective, the platform allows developers to create “solutions” by combining reusable prebuilt components (referred to as “apps”) which are offered by the platform provider –as commonly happens in PaaS platforms, but also created and offered by independent third-parties.

The ability to construct applications in this way —i.e. through the reuse of building blocks provided by third-parties within a platform's ecosystem, represents a major evolutionary force in the software industry today. It also creates a need for much more advanced mechanisms for quality assurance. The openness and complexity of the environment that emerges makes stability and reliability much harder to guarantee and calls for a rigorous approach to governance [121], [22]. The CAST project undertook to research this important requirement and to develop a suitable governance support system.

Following an introduction to the CAST project and the fundamentals of the CAST platform we discuss the governance requirements emerging in this context, the ecosystem actors in the governance process and their roles and presented examples of governance policies. We then introduce the governance support system that was created in the CAST project to address those requirements.

The approach taken in the CAST project was to develop a special-purpose registry & repository system that complements the CAST platform runtime environment. The system was developed through extensions and customisations on top of the open-source WSO2 Governance Registry platform [178]. Drawing examples from the set of policies produced in the scope of project CAST we illustrate how the governance support system developed by the project supports policy definition, data extraction and policy evaluation.

We then demonstrate an alternative approach to policy-based governance for the CAST platform based on the new framework for governance support system development that is put forward by our research. We discuss how policy definition, data extraction and policy evaluation are supported by applying the new design approach on the same set of policies. This helps to highlight that a governance support system based on PROBE is not only sufficient to meet the requirements of policy-governance in a CAST-powered ecosystem, but has several advantages over the original design approach.

The advantages of the PROBE framework are made concrete and demonstrated in greater depth through a structured comparative evaluation of the two design approaches. To evaluate the two approaches we adopt a scenario-based comparison methodology inspired by Bengtsson's method for architecture-level modifiability analysis (ALMA) [182]. We use four indicative "change scenarios" to compare the two design approaches in terms of how well they support evolvability and manageability of the governance process. The ecosystem governance requirements that were analysed in section 3.4 provide the benchmark to then assess the two design approaches.

The defining questions for the evaluation of the two alternative design approaches are: Does the design approach promote effectiveness and efficiency in the internal management of the governance process by the different governance roles (policy provider, data provider, policy evaluator)? Does it promote evolvability of the

governance process by each of the roles individually? Does it promote both goals at the same time?

A design approach based on the PROBE framework is shown to achieve better separation of concerns between the three main roles manifesting in a policy-based ecosystem governance context. It is also shown to significantly improve how policy definition, data extraction and policy evaluation processes can be managed.

Based on the analysis, we argue that PROBE represents an equally feasible but much more suitable design approach for developing governance support systems compared to what contemporary tools can offer; a design approach that is natively suited to the task of supporting governance in a continuously evolving cloud service ecosystem.

The key takeaways from this chapter can be summarised as follows:

1. Project CAST was an industry-academia R&D effort that developed technology to facilitate the co-development of cloud-based enterprise software solutions by an ecosystem of software companies. We here use the governance requirements of that specific ecosystem to compare two alternative approaches to developing governance support systems. The first approach is the one adopted by the consortium of research project CAST and the second is the one made possible by implementing the PROBE framework.
2. For each of the two approaches, we describe how the governance support system implements policy definition, data extraction and policy evaluation. We also evaluate and compare the two systems using change scenarios, inspired by Bengtsson's method for architecture-level modifiability analysis (ALMA). The defining questions for the evaluation of each system are: Does it promote effectiveness and efficiency in the internal management of the governance functions assumed by different roles? Does it promote evolvability of the individual governance functions? Does it promote both at the same time?
3. In the governance support system developed following the first approach, as originally developed in the scope of project CAST, the place where policies are defined, where relevant data is extracted and where policy evaluation logic is applied coincide in the implementation of the same architectural component: policy-specific data validators. The lack of proper separation of concerns between different governance functions limits the ability of ecosystem partners to co-evolve and impairs operational efficiency.
4. Because of its foundation on the Web Ontology Language (OWL) standard and related Semantic Web technologies, the PROBE-based method of defining governance policies is readily equipped to support heterogeneity, distribution and continuous evolution. By virtue of OWL's declarative encoding style and its formal logic underpinnings PROBE also facilitates advanced automation in policy engineering tasks. Employing a Linked Data approach for the description

of governance subjects achieves a loose coupling between the resources being governed, the governance policies and the policy evaluation mechanisms. Governance functions are decoupled, interoperability is ensured, abstraction is raised and ambiguity is avoided, while operational efficiency is improved.

5. PROBE is natively suited to the task of supporting governance in a dynamic cloud service ecosystem. But this comes with a cost: higher learning curve for engineers and an additional layer of technology that introduces complexity and room for error.

Chapter 9

Conclusions

9 Conclusions

9.1 Introduction

There are three fundamental ideas that underpin this research and its contributions.

> Idea #1. Creating software systems to support governance processes in a dynamic cloud service ecosystem is a challenge of enabling decentralised and distributed collaboration between networked organisations.
>
> Idea #2. In addressing the challenge of heterogeneity, distribution and continuous evolution, governance support systems need to ensure interoperability between ecosystem partners and separation of concerns between the functions of the policy provider, the data provider and the policy evaluator.
>
> Idea #3. One feasible and useful way to achieve the kind of interoperability and separation of concerns required in cloud service ecosystems governance, with present day technology, is through a software architecture that embodies Linked Data principles and Semantic Web standards.

The sections that follow unfold these fundamental ideas and summarise the research work that was carried out and the results obtained. We discuss the significance of the research and bring the dissertation to a close with an overview of limitations and directions for further work.

9.2 Synopsis

9.2.1 The challenge of ecosystem governance

Increasingly, cloud application platforms follow an open architecture which allows third-parties to enrich the platform's capabilities with their own add-ons (section 2.2.3). This model of collaboration represents a novel form of software product co-development (section 2.3.1). Software platforms that facilitate co-development relationships between different organisations foster the creation of environments best characterised as software ecosystems (section 2.3.2). Cloud service ecosystems can be seen as a special class of software ecosystems (section 2.3.3).

The value of a cloud service ecosystem increases exponentially with more users and more complementary services. But in software, size and diversity are at odds with reliability. The cause of this tension is complexity. To manage complexity, ecosystem partners need to be able to exercise control over developments in the ecosystem that

may affect them, such as the introduction of a new service, a change to the characteristics of an existing service, or a change to how services are assembled. This is a challenge of governance.

We define governance in cloud service ecosystems as the process and the supporting systems for defining and enforcing policies to control the creation, provision and consumption of cloud services by different ecosystem partners (section 2.4.2). In relation to the main body of literature on the wider topic of software ecosystem governance, our definition extends beyond the single viewpoint of the ecosystem coordinator to incorporate the governance requirements of all participants in the ecosystem (section 2.4.3).

9.2.2 Requirements thinking for governance support systems

Governance in a cloud service ecosystem is effected through a collaborative process that may span multiple networked organisational units and enterprises. The entities that participate in a governance process assume one or more of three fundamental roles: policy provider, data provider, or policy evaluator (section 3.3.1). A governance process may engage more than one entity in the same type of role (e.g. several different entities may act as policy providers in the same process). Moreover, a single governance process may engage the same entity in more than one role simultaneously.

Each governance process role is associated with a distinct set of concerns. These revolve around manageability and evolvability of the individual role's function. That is, the function of policy provision, data provision or policy evaluation (section 3.3.3).

Because of the different concerns associated with each governance process role, the entities that assume the relevant roles exhibit different rates of change and different types of change over the lifetime of the governance process. Therefore, in designing a software system to support governance processes in a cloud service ecosystem, we need to achieve separation of concerns between the three governance process functions (section 3.4.1). Decoupling governance policies, governed resources, and policy evaluation engines allows ecosystem partners to manage their internal governance processes in a more efficient way, while they cooperate, coevolve and innovate along with the ecosystem.

9.2.3 The PROBE framework

Examining the state-of-the-art governance technology platforms which are available to the software industry today reveals a gap between the type of requirements they were originally designed to meet and the type of needs emerging in the context of cloud service ecosystems (section 4.2.2). By virtue of its nature as a distributed and collaborative process, supporting governance in this new context requires systems to

enable networked collaboration, interoperability, and to guarantee a higher level of operational efficiency (section 4.2.3).

The thesis explored by this research is that governance support systems that satisfy the advanced requirements of cloud service ecosystems are both feasible and useful to realise on the basis of Linked Data principles and Semantic Web standards (section 4.4). The insight that underlies this thesis is that semantic technologies of this kind have already been shown to provide successful solutions in the related problem domains of inter-enterprise interoperability and policy engineering, and there are lessons learnt which can be readily transferred (section 4.3).

Heterogeneity, distribution and continuous evolution are the fundamental characteristics of the web. Semantic Web standards and Linked Data principles have been designed on that foundation. These same characteristics are also fundamental properties of governance processes in cloud service ecosystems. The challenge in designing a software system architecture to support governance in a cloud service ecosystem is a challenge of coping with heterogeneity, distribution and continuous evolution.

As an additional benefit, beyond the capability to enable networked collaboration, Semantic Web standards can also guarantee the higher level of operational efficiency that ecosystem governance processes require. By virtue of the formal semantics, modelling abstraction and standards-based interfacing embodied by the standards, the design of governance support systems can benefit from improved reusability, maintainability, traceability and agility.

Our proposed framework architecture for policy-driven governance in cloud service ecosystems (PROBE) comprises four core components: a shared governance ontology; a repository of ontology-based policy definitions; mechanisms to generate ontology-based resource descriptions; a governance policy evaluation engine (section 4.5).

9.2.4 Realising the framework

Governance policies and governed resources are heterogeneous, physically distributed and under multiple different ownership domains. It is therefore imperative to standardise formats for data exchange. For this reason, we describe the conceptualisation and representation of an ontology that serves as a shared ecosystem vocabulary to describe governed software resources, and at the same time also provides the vocabulary to define the necessary types of governance policies (section 5.2).

The ontology-based policy representation method is sufficiently expressive to allow describing diverse forms of cloud service resources and policies, covering governance objectives ranging from strategy to operations, and descriptions ranging from pricing models to lifecycle transitions. It is also sufficiently expressive to represent both types

of governance policies (process and resource governance), as well as both positive and negative formulation of constraints (section 5.3).

Because of its foundation on the Web Ontology Language (OWL) standard and related Semantic Web technologies, the proposed method of defining governance policies is readily equipped to support not only interoperability but also heterogeneity, distribution and continuous evolution (section 4.3.2). It is natively suited to support the type of networked collaboration found in cloud service ecosystem governance. It allows decoupling governance functions by offering a way to describe the policy conditions separately from the governance subjects and the policy evaluation logic. It ensures interoperability by offering a platform-agnostic way for ecosystem partners to exchange/share policies and data over the internet. It increases abstraction, by allowing ecosystem partners to bridge their terminology spaces to a common ecosystem-level vocabulary.

Moreover, by virtue of OWL's declarative encoding style and its formal logic underpinnings, our proposed method also facilitates advanced automation in policy engineering tasks, such as traceability of logical dependencies between policies, detection of contradictions with other policies and debugging of complex policy logic (e.g. through satisfiability tests). More generally, the unambiguous interpretation and automated reasoning capabilities afforded by OWL's formal semantics fulfils the need of increased operational efficiency - through improved maintainability, reusability, traceability and overall agility.

Employing a Linked Data approach for the description of governance subjects achieves a loose coupling between the governance policies, the resources being governed and the policy evaluation mechanisms. Applying Linked Data principles means that governance data providers do the following: (a) use URIs as names for governed resources, (b) use HTTP URIs so that entities acting as policy evaluators can look up those names, (c) provide information about the governed resources in RDF when a URI is looked up with a SPARQL query, (d) include links to other URIs in the information returned to a SPARQL query, so that ecosystem partners can further discover more useful information (section 6.2).

The abstract data description framework provided by RDF allows structured and semi-structured data in different ownership domains to be easily exposed and shared across organisational boundaries and heterogeneous applications. For this reason, RDF provides a viable foundation to describe and to share descriptions of governed resources in the context of a cloud services ecosystem. These descriptions can be automatically derived from their native data sources through a process that is commonly referred to as "triplification" (section 6.3). Another benefit of using the RDF standard is the wide range of existing tools available to support this process.

There is a great degree of heterogeneity in the native data sources, but RDF and SPARQL allow us to abstract over the differences and provide a common description

layer for all resources associated with any governed resource in a software ecosystem. The usage scenarios for the Linked Data which is produced through this process can include much more than just governance policy enforcement (section 6.3).

Checking the resulting ontology-based RDF descriptions of resources against governance policies is a task which can be approached as two different kinds of computational problem: a problem of integrity constraint validation on ontology objects, or a problem of ontology object classification (section 7.2). In this research we have chosen to explore the DL classification-based approach. We show how OWL's open-world assumption (OWA) and absence of a unique name assumption (UNA) prevent us from directly utilising a DL reasoner's object classification service for data validation, and present a solution in the form of an algorithm for local closure axiom generation (section 7.3).

The strength of the algorithm is in its generality and reusability. The approach relieves the ecosystem partners that function as policy evaluators from the need to develop and maintain a custom semantic policy evaluation engine. Instead, they can use a standard OWL DL reasoner in combination with our generic local closure axiom algorithm. The resulting policy evaluation mechanism is natively suited to support networked collaboration and is generic enough to cover all the different types of governance policies and governed resource descriptions in the ecosystem.

9.2.5 Evaluating the framework

To evaluate the PROBE framework we compare the approach it makes possible to the approach of research project CAST, where a governance support system for a cloud service ecosystem was built on top of a popular governance technology platform.

For each approach, we first describe how the respective governance support system implements policy definition, data extraction and policy evaluation and reflect on the effectiveness and efficiency of each approach (sections 8.4 and 8.5). We then evaluate and compare the two governance support systems using change scenarios (section 8.6). The defining questions for the evaluation of each system are: Does it promote effectiveness and efficiency in the internal management of the governance functions assumed by different roles? Does it promote evolvability of the individual governance functions? Does it promote both objectives simultaneously?

In the governance support system developed in the scope of project CAST, lack of proper separation of concerns between different governance functions limits the ability of ecosystem partners to co-evolve, and reliance on purpose-built Java policy validators for representing and enforcing policies hinders operational efficiency. The design approach put forward by PROBE is shown to have several advantages: governance functions are decoupled and interoperability is ensured, abstraction is raised while ambiguity is avoided, and operational efficiency is significantly improved.

9.3 In support of the thesis

The author's thesis is that the advanced requirements for governance support systems posed by cloud service ecosystems can be successfully met with an architecture framework that integrates Semantic Web technologies and Linked Data principles.

This thesis has been strongly supported by the successful development and evaluation of the PROBE framework for policy-driven governance in cloud service ecosystems. The framework is introduced in chapter 4 and comprises four components:

1. A governance ontology encoded in OWL-DL which serves as shared vocabulary between ecosystem partners for policy definition and data description (chapter 5).
2. A method for ontology-based policy definition whereby process and resource governance policies are formulated in positive or negative form (chapter 5).
3. A method for creating descriptions of governed resources and sharing them between ecosystem partners based on RDF (chapter 6).
4. A method and prototype system for evaluating governance policies against governed resource descriptions based on DL reasoning (chapter 7).

The feasibility of the PROBE framework is demonstrated in chapters 5 to 7 using examples from an industrial case study on cloud service ecosystem governance originating from the CAST project. The usefulness of the framework is demonstrated in chapter 8 through a comparative evaluation based on the same case study.

In the following section we will summarise the work carried out in the scope of this research and outline the results obtained.

9.4 Research process and results

The work carried out in the scope of this research can be divided in the stages of problem domain analysis and solution domain research and development. The order in which the research work is outlined below reflects thematic structure rather than chronological order, or order of importance of results.

9.4.1 Problem domain analysis

Research on the subject of this dissertation initiated with a study of academic and technical literature in the problem domain: service-oriented computing, cloud computing models, characteristics of PaaS offerings, models of software co-development and third-party platform extensions, software platform ecosystems, service governance, software ecosystem governance and governance support systems. This led to identifying software ecosystems as an emerging transformational

phenomenon in the software industry and a high-potential field for software engineering research. It also led to identifying the role of governance control mechanisms in cloud service ecosystems as an emerging subtopic of critical importance for ecosystems and practical value for the software industry. The most relevant learnings from this literature research are summarised in chapter 2.

The author had the opportunity to be part of the team working on research project CAST, an EU-sponsored industry-academia research effort that developed technology to support co-development of cloud-based business software by ecosystems of software companies [3],[22]. CAST (Enabling customisation of SaaS applications by third parties[52]) was coordinated by CAS Software AG which specialises in cloud CRM platforms. Other partners included third-party ISVs from the ecosystem of CAS Software, and the South-East European Research Centre of the University of Sheffield as academic partner. In the scope of CAST the author analysed the governance requirements for the CAST platform ecosystem and captured a dataset of 37 governance policies whose characteristics and common patterns were studied.

> Result #1: This led to conceptualising that governance policies can be abstracted to the level of either process governance or resource governance, and formulated accordingly for lifecycle management and artefact validation. This idea is introduced in chapter 2 and discussed in more detail in chapters 4 and 5.

Following requirements analysis the author led the development of a purpose-built governance support system for the CAST service ecosystem based on WSO2 Governance Registry platform. An analysis of the architecture of alternative governance support systems that could be used as the foundation to build on, such as MuleSoft Galaxy, had previously been conducted. Some insights from this analysis are briefly presented in chapter 4 (section 4.2). After the development of the CAST governance support system was completed the project team including the author evaluated the strengths and limitations of the resulting system.

A first key observation from this evaluation relates to the underlying architecture of the governance support system which is provided by the WSO2 Governance Registry platform. The observation is that it cannot support the ecosystem coordinator in evaluating policies defined by third-parties within the ecosystem. The architecture of the governance support system does not allow decoupling the functions of different ecosystem partners who participate in the governance process. Consequently it prevents them from being able to evolve independently of each other. There is a fundamental assumption as to the governance process which is natively supported by the system. The assumption is that the actor who is evaluating and enforcing the governance policies is the same actor who defines these policies in the first instance, and who also owns all the relevant resource descriptions. This may be true for governance policies which are defined and enforced by the ecosystem coordinator

[52] http://seerc.org/projects/cast/

based on resource descriptions that are locally available, but as discussed in chapter 3, this is far from the only governance scenario possible in a cloud service ecosystem.

The author studied these limitations and drew requirements from descriptions of more complex ecosystems with distributed governance processes. Part of this research was carried out in the requirements analysis phase of another, larger-scale EU-sponsored research project which included software companies SAP, CAS Software and SingularLogic (Broker@Cloud: Continuous Quality Assurance and Optimisation for Cloud Service Brokers [53,54]). This analysis of ecosystem scenarios provided inspiration and examples to define the synthesised ecosystem governance scenarios which are presented in chapter 3 (section 3.2).

> Result #2: This led to conceptualising that there can be three distinct roles at play in a cloud service ecosystem governance process (the policy provider, the data provider and the policy evaluator); that these can be physically distributed among several different actors who are ecosystem partners; that the three roles have different governance concerns; and that the concerns of each of them need to be facilitated not only in isolation but in combination with the rest of the governance process participants. This separation of concerns gives rise to new requirements in developing governance support systems. To meet these requirements we need to allow decoupling of governance policies, resource descriptions and policy evaluation engines. This concept is introduced in chapter 3 and relates to one of the primary contributions of this research.

Another key observation resulting from the evaluation of the CAST ecosystem governance support system relates to the system's effectiveness in supporting policy representation, analysis and evaluation. In more general terms, it relates to its effectiveness for policy engineering. A most fundamental limitation is found in the way in which the system supports the definition and enforcement of policies. The policy rules are encoded in an imperative manner, directly in Java, and as part of the same code that checks the data for violations. This has many negative side-effects with respect to policy maintainability, verifiability, interoperability, reusability and overall governance agility.

> Result #3. The two key observations as above led to demonstrating that state-of-the-art service governance support systems follow certain design assumptions which do not meet the evolved governance requirements of cloud service ecosystems. Their architecture limits how well they can support the management of the governance functions from the individual perspective of the policy provider, the data provider and the policy evaluator, and their individual evolvability requirements. These limitations were briefly introduced in chapter 4 (section 4.2) and discussed in more depth in chapter 8.

[53] http://cordis.europa.eu/project/rcn/105609_en.html
[54] http://www.broker-cloud.eu/

> Result #4. As a corollary, results #2 and #3 give rise to new software engineering research questions under the theme of software ecosystems governance. What is the way in which governance support systems should evolve to address the requirements of distributed governance processes in a cloud service ecosystem? What would be a good basis to build on, to achieve this evolution? The value of this result lies in opening up a new research theme with the potential to transfer knowledge and solutions from a multitude of software engineering research streams. This outcome is discussed in chapter 4.

9.4.2 Solution development

Result #4 provides the motivation to present the author's thesis, which is that governance support systems that satisfy the evolved requirements of cloud service ecosystems are both feasible and useful to develop with an architecture framework that integrates Semantic Web technologies and Linked Data principles.

The hypothesis underlying this thesis is that ontology-driven information systems engineering, Semantic Web technologies and Linked Data principles have already been shown to successfully provide solutions with analogous properties in other closely related problem domains.

Research into the literature and the technology relevant to the solution domain included: knowledge representation and reasoning, ontological modelling, description logics, DL reasoning, datalog rules, Semantic Web standards, Linked Data application architectures, policy modelling languages, ontology-policy-based management architectures, ontology-driven modularisation of information systems. The potential of 'semantic technology' to provide a new foundation for the development of governance support systems suited to the task was discussed in chapter 4 (section 4.3).

> Result #5. On this basis the author developed a conceptual framework titled PROBE (policy-driven governance in cloud service ecosystems) integrating Semantic Web technologies and Linked Data principles to guide the design and development of governance support systems meeting the requirements stated previously. This result is discussed in chapter 4 and represents a key contribution of this research.

To demonstrate the *feasibility* of the abstract PROBE framework, its components were developed in concrete form using the governance policies from project CAST as development use cases and test cases. The framework was shown to be able to support the full set of policies.

> Result #6. A governance ontology encoded in OWL-DL which serves as shared vocabulary between ecosystem partners for policy definition and data description. The ontology comprises (i) a core of generic domain concepts which are independent of any concrete cloud service ecosystem, (ii) modelling

constructs specific to the CAST platform and (iii) definitions of concrete policies from CAST platform examples. This result is presented in chapter 5 (section 5.2).

Result #7. A method for ontology-based policy definition whereby process and resource governance policies are formulated in positive or negative form. The description of the method is accompanied by several examples of positive/negative policy formulation for process and resource policies expressed as OWL class axioms and SWRL rules. This result is presented in chapter 5 (section 5.3).

Result #8. A method for creating RDF descriptions of governed resources and sharing them between ecosystem partners. We build on existing work by others in the field of Linked Data applications and describe an approach that combines transformational mappings against the governance ontology, dynamic on-demand generation of RDF triples and SPARQL-based access. This result is discussed in chapter 6 (section 6.3).

Result #9. A method and prototype system for evaluating governance policies against governed resource descriptions based on OWL-DL reasoning. Checking ontology-based descriptions of ecosystem resources against governance policies is a task which can generally be viewed as two different kinds of computational problem: a problem of integrity constraint validation on ontology objects, or a problem of ontology object classification. We presented a method and implemented a prototype in Java for policy evaluation based on DL reasoning which allows distributed and independent policy evaluators to use a common policy conformance checking infrastructure. One that is generic enough to cover all the different types of governance policies and governance resource descriptions in the ecosystem. This includes a novel algorithm for local closure axiom generation that allows an open-world reasoning engine such as a standard OWL-DL reasoner to operate in a closed-world setting and produce the desired inferences so as to successfully check policy conformance. This result is discussed in chapter 7 (section 7.3).

To demonstrate the *usefulness* of the PROBE framework, the author compared the original design approach taken in research project CAST to create a governance support system based on the open-source governance platform by WSO2, to the approach made possible by adopting the PROBE framework.

Result #10. A comparative evaluation of alternative governance support system architecture approaches using change scenarios, which demonstrates the advantages of the PROBE framework over the solutions afforded by state-of-the-art governance support systems. The first design approach evaluated is the one originally adopted by project CAST for supporting governance on the CAST cloud application platform. The second design

approach is the one adopted by the PROBE framework as introduced in chapters 5 to 7 of this dissertation. We discuss how policy definition, data extraction and policy evaluation are supported by applying the new design approach of PROBE on the same set of policies from the CAST project. We examine the two approaches from the perspective of the different roles involved in the ecosystem governance process: policy provider, data provider, and policy evaluator, using a scenario-based comparison methodology. Change-scenarios are employed to evaluate the two design approaches in terms of how they support evolvability and manageability of the governance process. This helps to highlight that a governance support system based on PROBE is not only sufficient to meet the requirements of governance in a CAST platform ecosystem, but has several advantages over the original design approach. This result is discussed in chapter 8 (section 8.6).

9.4.3 Summary of results

The table below summarises the research results in order of presentation.

Result	Description	Chapter
R1.	Conceptualising that governance policies can be abstracted to either process governance or resource governance levels	4, 5
R2.	Conceptualising the three roles that manifest in a cloud service ecosystem governance process, and their associated concerns	3
R3.	Demonstrating limitations of state of the art governance support systems for governance in cloud service ecosystems	4, 8
R4.	New research questions: How should the design of governance support system evolve? What is the basis to achieve this evolution?	4
R5.	Conceptualising the PROBE framework	4
R6.	Developing the governance ontology	5
R7.	Describing a method for ontology-based policy definition	5
R8.	Describing a method for RDF-based data description	6
R9.	Describing a method and developing a prototype system for policy evaluation	7
R10.	Comparative evaluation: CAST vs PROBE design approach	8

Table 53. Summary of research results

9.5 Significance of results and contributions

Because of the growing importance of co-development in software products and services, governance in cloud service ecosystems —and software ecosystems in

general— is an emerging research theme of high impact and immediate practical value for the software industry.

The intended contribution of this research work to computer science and to the cross-disciplinary field of software ecosystems research is threefold:

1. Contribution #1. Furthering our understanding of the problem domain of governance in cloud service ecosystems.
2. Contribution #2. Presenting a conceptual model to analyse the interactions between ecosystem partners in a governance process and the requirements posed for governance support systems.
3. Contribution #3. Creating a feasible and useful solution framework that advances the state of the art in engineering governance support systems for software ecosystems.

9.5.1 Furthering our understanding of governance

This research contributes towards filling a gap in the software ecosystems research literature, since the vast majority of works on ecosystem governance have so far focused on analysing or proposing models to frame governance decision-making instead of models to engineer governance support systems. Focus so far has been on studying strategic governance policy making from a management viewpoint, rather than operational governance control and policy enforcement from an engineering viewpoint, which is the focus of this work.

Moreover, the body of literature on the subject of software ecosystem governance has so far focused, almost exclusively, on governance of the ecosystem from the viewpoint of the keystone partner. However, it is not only the keystone partner that has governance objectives. Governance processes inside an ecosystem may involve multiple distributed ecosystem partners who assume different types of roles and need to interoperate and collaborate as a network.

Accordingly, our definition of governance in cloud service ecosystems extends beyond the narrow viewpoint of the ecosystem coordinator to incorporate the wider governance requirements of all participants in the ecosystem. From the viewpoint of the ecosystem coordinator, governance is about ensuring that the introduction of new services – or the modification of existing ones – will not create a negative impact on the ecosystem's stability and reliability. From the viewpoint of ecosystem participants (third-party developers and end-users) governance is about ensuring that the services they consume or deliver operate as required on a continuous basis.

This research brings attention to the aspects of decentralisation, distribution and networked collaboration in ecosystem governance. We put forward the key idea that creating software systems to support governance processes in a dynamic cloud service

ecosystem is actually a challenge of enabling decentralised and distributed collaboration between networked organisations that continuously evolve.

To the best of the author's knowledge, this is the first time software ecosystem governance is framed in such a way.

9.5.2 Providing a conceptual model for requirements thinking

Framing the challenge of governance in a cloud service ecosystem as a challenge of heterogeneity, distribution and continuous evolution, leads to a new way of thinking about the requirements that governance support systems must fulfil.

We introduce the key idea that to meet the challenge of governance in a cloud service ecosystem, governance support systems need to achieve clear separation of concerns between three core functions found in every governance process: the functions of the policy provider, the data provider and the policy evaluator.

Associated with each of these functions is a distinct set of concerns and goals, which the respective stakeholder(s) would like to see satisfied. Some are role-level concerns which are local in scope and mainly relate to manageability of the individual function from the partner's perspective. Others are ecosystem-level concerns which are global in scope, and mainly relate to evolvability and decentralisation of the process. Because of the different concerns associated with each function, the ecosystem partners that assume the relevant process roles exhibit different rates of change and different types of change over the lifetime of the governance process.

Therefore, in designing a software system to support collaborative governance processes, we need to ensure that the role-driven concerns of the different entities engaged in the process are independently addressed and simultaneously satisfied. And at the same time, we need to ensure that governance support systems facilitate interoperability between the ecosystem partners that function in different roles.

To the best of our knowledge this is the first time that this designation of different governance process roles and associated concerns is made in research literature on software ecosystems or governance.

9.5.3 Delivering a feasible and useful solution framework

Formulating the problem space as described above leads to observing the opportunity of transferring best practices and technology from existing solution spaces that address problems of similar nature.

We introduce the key idea that one possible way to achieve the sought interoperability between ecosystem partners and separation of concerns between governance functions, with present day technology, is through a software architecture that embodies Linked

Data principles and Semantic Web standards. This idea is the actual research thesis that this work sets out to investigate in depth, by proposing a novel conceptual software architecture framework, showing how this framework can be realised with existing technology standards, and evaluating the results in an actual cloud service ecosystem case study.

The underlying observation is that heterogeneity, distribution and continuous evolution are fundamental characteristics of the web, and these same characteristics provided the motivation for creating Semantic Web standards and Linked Data principles. But these characteristics are also fundamental properties of governance processes in cloud service ecosystems, making Semantic Web standards and Linked Data principles a good candidate to offer solutions in this space.

As mentioned earlier, beyond the capability to enable networked collaboration, the added benefit of Semantic Web standards is that they can also guarantee the higher level of operational efficiency that ecosystem governance processes require. By virtue of the formal semantics, modelling abstraction and standards-based interfacing embodied by the standards, the design of governance support systems can benefit from improved reusability, maintainability, traceability and agility.

A design approach based on the PROBE framework is shown to achieve better separation of concerns between the three main roles manifesting in a cloud service ecosystem governance context. It is also shown to significantly improve how policy definition, data representation and policy evaluation processes can be managed. Overall, the PROBE framework provides an approach that is natively suited to the task of supporting governance in a continuously evolving cloud service ecosystem.

To the best of our knowledge this is the first time that such a software framework is proposed in the scope of cloud service ecosystems governance, or software ecosystem governance in general.

Implicit in this thesis is the view that in twenty years from now, the technology that will be most suitable to achieve the goals described here will most probably be quite different. Web standards will have evolved and new software architecture patterns will be available to facilitate networked collaboration between software ecosystem partners, along with separation of concerns between distributed governance process functions. However, the principles guiding the requirements that governance support systems need to fulfil will fundamentally remain the same.

9.6 Limitations and further work

9.6.1 Further case studies of governance in cloud service ecosystems

In the course of this research the author was fortunate to have worked on two large-scale research projects which involved collaboration between software industry and academia. Both projects, CAST and Broker@Cloud, included the objective to investigate governance in cloud service ecosystems from a software engineering research viewpoint. Both provided valuable usage scenarios and concrete governance control examples in the form of policies.

Access to more ecosystem case studies would have been helpful to add to our understanding of the problem domain and to further evaluate the feasibility and usefulness of the PROBE framework. We have been able to demonstrate adequacy of the PROBE framework against the requirements formed by studying this case study material but we cannot, and do not, claim generality. The CAST project in specific, which provided the ecosystem governance policies used in this work, focused its attention to study and facilitate ecosystem scenarios where policy evaluation is performed centrally by the CAST platform operator, unlike other ecosystem settings where policy evaluation can be carried out by different ecosystem partners.

CAST took a coordinator-centric perspective on policy evaluation, although it opened up policy definition as something other third-parties could do. CAST did not consider consumers of cloud services (end-users) as policy providers, data providers or policy evaluators. This area was later investigated further in the scope of a project Broker@Cloud where new scenarios where explored. This exploration showed that CAST does not represent the most complex ecosystem governance environment possible. This is also illustrated by the exemplifying scenarios presented in chapter 3.

To deepen our understanding of the problem of governance in cloud service ecosystems this research needs to be expanded with additional case studies of governance requirements in cloud service ecosystems. A related research extension is a comparative evaluation of a PROBE-based governance support system against other governance support systems as implemented by cloud service ecosystems, using the same or an expanded set of change scenarios.

9.6.2 Comparison to other commercial governance technology platforms

One of the outcomes from this research was demonstrating the limitations of state of the art governance support systems for governance in cloud service ecosystems (result #2). This was based on a survey and technical analysis of the governance platforms that were available for the author to access at the time of the research. The analysis was focused on commercial governance technology platforms which are also open-source and can be accessed free of charge.

Other commercial solutions in the market were only studied through the documentation that was publicly available. It is possible that there are other commercial solutions for governance in the market which provide better policy engineering support by separating policy definition from policy enforcement logic. We therefore do not claim generality for the observations made. An extension of this research could involve more governance technology platforms to be evaluated against cloud service ecosystem governance requirements.

9.6.3 Alignment of the governance ontology to Linked-USDL

The governance ontology (result #6) comprises only a small core of platform-independent governance concepts. The primary focus in developing this ontology wasn't to provide a foundational domain ontology for cloud service ecosystems governance such as Dublin Core[55] or GoodRelations[56], although the resulting ontology could be extended to that direction. The ontology was developed with the purpose to demonstrate the feasibility of the respective component in the PROBE framework which is done through showing that OWL-DL as a modelling formalism is sufficiently expressive to allow representing the full set of artefact validation and lifecycle management policies from CAST.

Due to time limitations, no attempt was made to align the ontology vocabulary with other domain ontologies such as Linked-USDL[57][183]. Nevertheless, the ontology presented could easily be extended towards to achieve this. Given the alignment with the applications that Linked-USDL is intended to support, it is worthwhile to explore connecting the governance ontology vocabulary to the core ontology provided by USDL (Linked USDL Core). This is both interesting and useful to pursue because of the benefits it could have from an interoperability perspective. Describing governed resources based on such a vocabulary would open up many different uses for the data. It would also benefit reusability of the descriptions and would allow using the Linked-USDL authoring tools available.

9.6.4 Alternative policy evaluation approaches

Checking ontology-based descriptions of ecosystem resources against governance policies is a task which can generally be viewed as (at least) two different kinds of computational problem: a problem of integrity constraint validation on ontology objects, or a problem of ontology object classification. The ontology-based policy definition method (result# 7) and evaluation methodology (result #9) presented in this dissertation adopt the approach of object classification with DL reasoning.

[55] http://dublincore.org/
[56] http://www.heppnetz.de/projects/goodrelations/
[57] https://linked-usdl.org/

An interesting further research project would be to explore the relative advantages and disadvantages of the alternative approach that was only briefly introduced. Assuming the alternative approach based on query answering also proves feasible, both approaches would represent options to instantiate the PROBE framework.

9.6.5 Data interlinking and sharing infrastructure

The RDF-based method for the description and sharing of governed ecosystem resource descriptions (#result 8) is limited to providing guidelines for the description of resources, and guidelines for implementing the data extraction and sharing architecture using third-party technologies, such as RDB to RDF and RDF triple stores.

This research work did not include any implementation of a prototype for governance data extraction and sharing. This was not deemed as a high-impact research question as it does not put the feasibility of the PROBE framework in question. It would however be interesting as a further research project to develop more concrete guidelines or even a reusable software framework that others could build on.

9.6.6 PROBE framework integration toolkit

PROBE is put forward as a framework to inform the design and development of governance support systems. The framework is not proposed as an alternative to using governance technology platforms such as WSO2 Governance Registry. Our design intention is not to reinvent the wheel but for the PROBE framework to constitute an additional architecture layer on top of such systems.

Integration with such platforms would be case-specific but could nevertheless be guided by a common integration framework and an event-driven architecture to allow easy integration. This was not developed as part of this research and could be a subject of further work with different objectives.

9.6.7 Application to other classes of software ecosystems

Cloud service ecosystems have been identified as a subclass of software ecosystems. This raises the question: Do other classes of software ecosystems present similar needs with regards to governance? Can the PROBE framework be successfully applied to improve governance support for other software ecosystems, such as mobile software platforms?

Our comparison between the extensibility architectures of mobile platforms versus cloud application platforms (not reported in this dissertation) suggests that the latter type of ecosystems offer much more advanced models of co-development at the present time, but the research questions are still of interest.

9.7 Publications by the author

The table below presents the most relevant publications by the author in chronological order, and links them to work presented in chapters of this dissertation.

Publication	**Related chapter**
Simons, A. J., Bratanis, K., Kourtesis, D., Paraskakis, I., Veloudis, S., Verginadis, Y., ... & Rossini, A. (2014, December). Advanced service brokerage capabilities as the catalyst for future cloud service ecosystems. In Proceedings of the 2nd International Workshop on CrossCloud Systems (p. 7). ACM.	2, 3, 4
Bratanis, K., & Kourtesis, D. (2014). Introducing Policy-Driven Governance and Service Level Failure Mitigation in Cloud Service Brokers: Challenges Ahead. In I. Brandi & F. Patrizi B. Benatallah S. N. A. Lomuscio (Ed.), Service-Oriented Computing, ICSOC 2013 Workshops and PhD Symposium. Lecture Notes in Computer Science (LNCS) (Vol. 8377). Springer Berlin / Heidelberg.	2, 3, 4
Kourtesis, D., Bratanis, K., Verginadis, Y., Friesen, A., Simons, A. J. H., Rossini, A., et al. (2014). Brokerage for Quality Assurance and Optimisation of Cloud Services: an Analysis of Key Requirements. In I. Brandi & F. Patrizi B. Benatallah S. N. A. Lomuscio (Ed.), Service-Oriented Computing, ICSOC 2013 Workshops and PhD Symposium. Lecture Notes in Computer Science (LNCS) (Vol. 8377). Springer Berlin / Heidelberg.	2, 3, 4
Bratanis, K., Kourtesis, D., Paraskakis, I., Verginadis, Y., Mentzas, G., Simons, A. J. H., et al. (2013). A research roadmap for bringing continuous quality assurance and optimization to enterprise cloud service brokers. In Proceedings of eChallenges 2013. Dublin, Ireland.	2, 3, 4
Kourtesis, D., & Bratanis, K. (2013). Towards Continuous Quality Assurance in Future Enterprise Cloud Service Brokers. In Proceedings of the 8th South East European Doctoral Student Conference (DSC 2013). Thessaloniki, Greece.	2, 3, 4
Kourtesis, D., Bratanis, K., Bibikas, D., & Paraskakis, I. (2012). Software Co-development in the Era of Cloud Application Platforms and Ecosystems: the Case of CAST. In Collaborative Networks in the Internet of Services – 13th IFIP WG 5.5 Working Conference on Virtual Enterprises (pp. 196–204). IFIP Advances in Information and Communication Technology, 380. Springer Boston.	2, 3, 8
Kourtesis, D., Bratanis, K., & Paraskakis, I. (2012). Continuous governance and quality control in a next-generation enterprise cloud application platform. IT Briefcase, (11 July 2012).	8
Kourtesis, D., Paraskakis, I., & Simons, A. J. H. (2012). Ontology-based framework for policy-driven governance in cloud application platforms. In Proceedings of the 7th International Conference on Formal Ontology in Information Systems (FOIS 2012). Graz, Austria.	4, 5, 6, 7
Kourtesis, D., Paraskakis, I., & Simons, A. J. H. (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 (ODISE 2012). Graz, Austria.	4, 5, 6, 7

Kourtesis, D., & Paraskakis, I. (2011). A Registry & Repository System Supporting Cloud Application Platform Governance. In ICSOC 2011 – Ninth International Conference on Service Oriented Computing (pp. 255–256). Lecture Notes in Computer Science, 7221. Springer Berlin / Heidelberg.	8
Kourtesis, D., & Paraskakis, I. (2011). Governance in Cloud Platforms for the Development and Deployment of Enterprise Applications. In IEEE CloudCom 2011 – 3rd IEEE International Conference on Cloud Computing Technology and Science.	3, 8
Kourtesis, D., Kuttruff, V., & Paraskakis, I. (2010). Optimising development and deployment of enterprise software applications on PaaS: The CAST project. In M. Cezon, & Y. Wolfsthal (Eds.), ServiceWave 2010 Workshops (pp. 14–25). Lecture Notes in Computer Science (LNCS), 6569/2011. Springer Berlin / Heidelberg.	2, 3, 8
Kourtesis, D., Paraskakis, I., & Simons, A. J. H. (2009). Semantic Web Technologies in Support of Service Oriented Architecture Governance. In Proceedings of the 4th South East European Doctoral Student Conference (DSC 2009) (pp. 418–425). Thessaloniki, Greece: South-East European Research Centre (SEERC).	2, 4
Kourtesis, D., & Paraskakis, I. (2009). Supporting Semantically Enhanced Web Service Discovery for Enterprise Application Integration. In G. Mentzas, & A. Friesen (Eds.), Semantic Enterprise Application Integration for Business Processes: Service-Oriented Frameworks. Hersley: IGI Global.	6, 7
Kourtesis, D., & Paraskakis, I. (2008). Combining SAWSDL, OWL-DL and UDDI for Semantically Enhanced Web Service Discovery. In S. Bechhofer, M. Hauswirth, J. Hoffmann, & M. Koubarakis (Eds.), The Semantic Web: Research and Applications (pp. 614–628). Lecture Notes in Computer Science (LNCS), 5021. Springer Berlin / Heidelberg.	6, 7

Table 54. Summary of related publications by the author

10 References

[1] N. G. Carr, "The end of corporate computing," *MIT Sloan Manag. Rev.*, vol. 46, no. 3, pp. 67–73, 2005.

[2] M. Skok, "Future of cloud computing—4th annual survey results." North Bridge Venture Partners, 2014.

[3] D. Kourtesis, K. Bratanis, D. Bibikas, and I. Paraskakis, "Software co-development in the era of cloud application platforms and ecosystems: The case of CAST," in *IFIP Advances in Information and Communication Technology*, 2012, vol. 380 AICT, pp. 196–204.

[4] G. K. Hanssen and T. Dybå, "Theoretical foundations of software ecosystems.," in *IWSECO@ ICSOB*, 2012, pp. 6–17.

[5] S. Jansen, A. Finkelstein, and S. Brinkkemper, "A sense of community: A research agenda for software ecosystems," in *31st International Conference on Software Engineering - Companion Volume*, 2009, pp. 187–190.

[6] A. Gawer and M. A. Cusumano, *Platform leadership: How Intel, Microsoft, and Cisco drive industry innovation*. Harvard Business School Press, 2002.

[7] J. Pan, "Software Reliability," *Dependable Embed. Syst.*, vol. 18, pp. 1–14, 1999.

[8] K. S. Lew, T. S. Dillon, and K. E. Forward, "Software complexity and its impact on software reliability," *IEEE Trans. Softw. Eng.*, vol. 14, no. 11, pp. 1645–1655, 1988.

[9] K. Manikas, "Revisiting software ecosystems Research: A longitudinal literature study," *J. Syst. Softw.*, vol. 117, pp. 84–103, Jul. 2016.

[10] A. J. H. Simons, K. Bratanis, D. Kourtesis, I. Paraskakis, S. Veloudis, Y. Verginadis, G. Mentzas, S. Braun, and A. Rossini, "Advanced service brokerage capabilities as the catalyst for future cloud service ecosystems," in *Proceedings of the 2nd International Workshop on CrossCloud Systems - CCB14*, 2014.

[11] D. Kourtesis, K. Bratanis, Y. Verginadis, A. Friesen, A. J. H. Simons, A. Rossini, A. Schwichtenberg, and P. Gouvas, "Brokerage for Quality Assurance and Optimisation of Cloud Services: An Analysis of Key Requirements," in *Lecture Notes in Computer Science*, Springer Science + Business Media, 2014, pp. 150–162.

[12] K. Bratanis and D. Kourtesis, "Introducing policy-driven governance and service level failure mitigation in cloud service brokers: Challenges ahead," in *Lecture Notes in Computer Science*, 2014, vol. 8377 LNCS, pp. 177–191.

[13] K. Bratanis, D. Kourtesis, I. Paraskakis, Y. Verginadis, G. Mentzas, A. J. H. Simons, A. Friesen, and S. Braun, "A research roadmap for bringing continuous quality assurance and optimization to enterprise cloud service brokers," in

Proceedings of eChallenges, 2013.

[14] G. Antoniou and F. Van Harmelen, *A semantic web primer*. MIT press, 2004.

[15] C. Bizer, T. Heath, and T. Berners-Lee, "Linked Data - The Story So Far," *Int. J. Semant. Web Inf. Syst.*, vol. 5, no. 3, pp. 1–22, 2009.

[16] M. Uschold, "Ontology-driven information systems: Past, present and future," in *Proceedings of the 2008 conference on Formal Ontology in Information Systems: Proceedings of the Fifth International Conference (FOIS 2008)*, 2008, pp. 3–18.

[17] G. Tonti, J. M. M. Bradshaw, R. Jeffers, R. Montanari, N. Suri, and A. Uszok, "Semantic Web Languages for Policy Representation and Reasoning: A Comparison of KAoS, Rei, and Ponder," in *Lecture Notes in Computer Science*, Springer Science & Business Media, 2003, pp. 419–437.

[18] A. Uszok, J. M. M. Bradshaw, M. Johnson, R. Jeffers, A. Tate, J. Dalton, and S. Aitken, "KAoS policy management for semantic Web services," *IEEE Intell. Syst.*, vol. 19, no. 4, pp. 32–41, Jul. 2004.

[19] T. Heath and C. Bizer, "Linked Data: Evolving the Web into a Global Data Space," *Synth. Lect. Semant. Web Theory Technol.*, vol. 1, no. 1, pp. 1–136, Feb. 2011.

[20] S. Staab and R. Studer, Eds., *Handbook on Ontologies*. Springer Nature, 2009.

[21] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, and others, "SWRL: A semantic web rule language combining OWL and RuleML," *W3C Memb. Submiss.*, vol. 21, p. 79, 2004.

[22] D. Kourtesis, V. Kuttruff, and I. Paraskakis, "Optimising development and deployment of enterprise software applications on PaaS: the CAST project," in *Lecture Notes in Computer Science*, 2010, vol. 6569 LNCS, pp. 14–25.

[23] L. Feigenbaum, G. T. Williams, K. G. Clark, and E. Torres, "SPARQL 1.1 Protocol," *Recomm. W3C, March*, 2013.

[24] M. A. Rappa, "The utility business model and the future of computing services," *IBM Syst. J.*, vol. 43, no. 1, pp. 32–42, 2004.

[25] R. W. Seidel, "Architects of the Information Society: Thirty-Five Years of the Laboratory for Computer Science at MIT by Simson L. Garfinkel, edited by Hal Abelson," *Endeavour*, vol. 24, no. 2, p. 94, Jun. 2000.

[26] D. F. Parkhill, "Challenge of the computer utility," 1966.

[27] A. T. Velte, T. J. Velte, R. C. Elsenpeter, and R. C. Elsenpeter, *Cloud computing: a practical approach*. McGraw-Hill New York, 2010.

[28] "Conversation with Eric Schmidt hosted by Danny Sullivan," in *Search Engine Strategies Conference*, 2006.

[29] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, “A break in the clouds,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, p. 50, Dec. 2008.

[30] N. Leavitt, “Is Cloud Computing Really Ready for Prime Time?,” *Computer (Long. Beach. Calif).*, vol. 42, no. 1, pp. 15–20, Jan. 2009.

[31] M. Armbrust, I. Stoica, M. Zaharia, A. Fox, R. Griffith, A. D. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, and A. Rabkin, “A view of cloud computing,” *Commun. ACM*, vol. 53, no. 4, p. 50, Apr. 2010.

[32] P. M. Mell and T. Grance, “The NIST definition of cloud computing,” National Institute of Standards and Technology (NIST), techreport, 2011.

[33] Y. Natis, “Introducing SaaS-Enabled Application Platforms: Features, Roles and Futures,” Gartner Inc, 2007.

[34] Y. Chou, “Cloud Computing Primer for IT Pros,” 2010. [Online]. Available: https://blogs.technet.microsoft.com/yungchou/2010/11/15/cloud-computing-primer-for-it-pros/.

[35] S. Jansen and E. Bloemendal, “Defining App Stores: The Role of Curated Marketplaces in Software Ecosystems,” in *Lecture Notes in Business Information Processing*, Springer Science & Business Media, 2013, pp. 195–206.

[36] H. Chesbrough and K. Schwartz, “Innovating business models with co-development partnerships,” *Res. Manag.*, vol. 50, no. 1, pp. 55–59, 2007.

[37] G. Büyüközkan and J. Arsenyan, “Collaborative product development: a literature overview,” *Prod. Plan. Control*, vol. 23, no. 1, pp. 47–66, Jan. 2012.

[38] CollabNet, “Enabling Open Collaboration with Development Partners,” 2007.

[39] G. Parker and M. Van Alstyne, “Managing platform ecosystems,” *ICIS 2008 Proc.*, p. 53, 2008.

[40] M. H. Meyer and R. Seliger, “Product platforms in software development,” *MIT Sloan Manag. Rev.*, vol. 40, no. 1, p. 61, 1998.

[41] D. S. Evans, “A Survey of the Economic Role of Software Platforms in Computer-based Industries,” *CESifo Econ. Stud.*, vol. 51, no. 2–3, pp. 189–224, Jan. 2005.

[42] D. G. Messerschmitt and C. Szyperski, “Software ecosystem: understanding an indispensable technology and industry,” *MIT Press Books*, vol. 1, 2003.

[43] S. Jansen and G. van Capelleveen, “Quality review and approval methods for extensions in software ecosystems,” in *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*, Edward Elgar Publishing, 2013, p. 187.

[44] J. Knodel and K. Manikas, “Towards a Typification of Software Ecosystems,” in *Lecture Notes in Business Information Processing*, Springer Science & Business Media, 2015, pp. 60–65.

[45] H.-B. Kittlaus and P. N. Clough, *Software product management and pricing: Key success factors for software organizations*. Springer Science & Business Media, 2008.

[46] J. Bosch and P. Bosch-Sijtsema, “From integration to composition: On the impact of software product lines, global development and ecosystems,” *J. Syst. Softw.*, vol. 83, no. 1, pp. 67–76, Jan. 2010.

[47] K. Manikas and K. M. Hansen, “Software ecosystems A systematic literature review,” *J. Syst. Softw.*, vol. 86, no. 5, pp. 1294–1306, May 2013.

[48] S. Jansen and M. A. Cusumano, “Defining software ecosystems: a survey of software platforms and business network governance.” Edward Elgar Publishing, pp. 13–28.

[49] J. F. Moore, *The death of competition: leadership and strategy in the age of business ecosystems*. HarperCollins Publishers, 1996.

[50] T. Rickmann, S. Wenzel, and K. Fischbach, “Software Ecosystem Orchestration: The Perspective of Complementors,” 2014.

[51] J. van Angeren, J. Kabbedijk, K. M. Popp, and S. Jansen, “Managing software ecosystems through partnering,” in *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*, Edward Elgar Publishing, 2013, pp. 85–102.

[52] M. Iansiti and R. Levien, *The keystone advantage: what the new dynamics of business ecosystems mean for strategy, innovation, and sustainability*. Harvard Business Press, 2004.

[53] T. R. Eisenmann, G. Parker, and M. Van Alstyne, “Opening Platforms: How, When and Why?,” *Platforms, Mark. Innov.*, pp. 131–162, 2008.

[54] S. Shane, *The handbook of technology and innovation management*. John Wiley & Sons, 2008.

[55] S. Jansen, S. Peeters, and S. Brinkkemper, “Software Ecosystems: From Software Product Management to Software Platform Management.,” in *IW-LCSP@ ICSOB*, 2013, pp. 5–18.

[56] L. Richardson and S. Ruby, *RESTful web services*. “ O’Reilly Media, Inc.,” 2008.

[57] “Origin and Etymology of Govern,” *Merriam Webster Dictionary*, 2016. .

[58] M. Bevir, *Governance: A Very Short Introduction*. Oxford University Press, 2012.

[59] W. A. Brown, G. Moore, and W. Tegan, “SOA governance—IBM’s approach, Effective governance through the IBM SOA Governance Management Method approach,” article, 2006.

[60] M. Afshar, M. Cincinatus, D. Hynes, K. Clugage, and V. Patwardhan, “SOA governance: Framework and best practices,” article, 2007.

[61] D. Krafzig, K. Banke, and D. Slama, *Enterprise SOA: service-oriented architecture best practices*. Prentice Hall Professional, 2005.

[62] D. S. Linthicum, “The Evolution of Cloud Service Governance,” *IEEE Cloud Comput.*, vol. 2, no. 6, pp. 86–89, Nov. 2015.

[63] A. Papageorgiou, S. Schulte, D. Schuller, M. Niemann, N. Repp, and R. Steinmetz, “Governance of a Service-Oriented Architecture for Environmental and Public Security,” in *Information Technologies in Environmental Engineering*, Springer Science & Business Media, 2009, pp. 39–52.

[64] O. Uludag, S. Hefele, and F. Matthes, “Platform and Ecosystem Governance,” *Digit. Mobil. Platforms Ecosyst.*, p. 1, 2016.

[65] A. Serebrenik and T. Mens, “Challenges in Software Ecosystems Research,” in *Proceedings of the 2015 European Conference on Software Architecture Workshops - ECSAW15*, 2015.

[66] R. Santos, O. Barbosa, and C. Alves, “Software ecosystems: trends and impacts on software engineering,” in *Software Engineering (SBES), 2012 26th Brazilian Symposium on*, 2012, pp. 206–210.

[67] A. Tiwana, *Platform Governance: Aligning Architecture, Governance, and Strategy*. Morgan Kaufmann, 2014.

[68] A. Tiwana, B. Konsynski, and A. A. Bush, “Platform Evolution: Coevolution of Platform Architecture, Governance, and Environmental Dynamics,” *Inf. Syst. Res.*, vol. 21, no. 4, pp. 675–687, Dec. 2010.

[69] M. Schreieck, M. Wiesche, and H. Krcmar, “Design and governance of plaform ecosystems -key concepts and issues for future research,” in *Twenty-Fourth European Conference on Information Systems (ECIS), İstanbul,Turkey*, 2016.

[70] J. Axelsson and M. Skoglund, “Quality assurance in software ecosystems: A systematic literature mapping and research agenda,” *J. Syst. Softw.*, vol. 114, pp. 69–81, Apr. 2016.

[71] J. van Angeren, C. Alves, and S. Jansen, “Can we ask you to collaborate? Analyzing app developer relationships in commercial platform ecosystems,” *J. Syst. Softw.*, vol. 113, pp. 430–445, Mar. 2016.

[72] E. A. Marks, *Service-oriented architecture (SOA) governance for the services driven enterprise*. John Wiley & Sons, 2008.

[73] L.-J. Zhang and Q. Zhou, “CCOA: Cloud computing open architecture,” in *Web*

Services, 2009. ICWS 2009. IEEE International Conference on, 2009, pp. 607–616.

[74] J. A. Estefan, K. Laskey, F. McCabe, and P. Thornton, "Reference Architecture Foundation for Service Oriented Architecture Version 1.0, OASIS Committee Draft 02, 14 October 2009." 2011.

[75] "Transforming Business: Optimizing the Business Outcomes of SOA," 2008.

[76] T. Manes, "The Registry and SOA Governance Market," 2007.

[77] P. Malinverno, D. C. Plummer, and G. Van Huizen, "Gartner magic quadrant for application services governance. Gartner Inc," misc, 2015.

[78] I. Kourtesis, D., Bratanis, K., & Paraskakis, "Continuous governance and quality control in a next-generation enterprise cloud application platform," *IT Briefcase*, 2012.

[79] S. Hyrynsalmi, A. Suominen, and M. Mäntymäki, "The influence of developer multi-homing on competition between software ecosystems," *J. Syst. Softw.*, vol. 111, pp. 119–127, Jan. 2016.

[80] E. W. Dijkstra, "On the role of scientific thought," *Selected writings on computing: a personal perspective*, Springer New York , pp. 60–66, 1982.

[81] D. L. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[82] T. Mens and M. Wermelinger, "Separation of concerns for software evolution," *J. Softw. Maint. Evol. Res. Pract.*, vol. 14, no. 5, pp. 311–315, 2002.

[83] H. Ossher and T. Peri, "Multi-dimensional separation of concerns and the hyperspace approach," *Software Architectures and Component Technology*, pp. 293–323, 2002.

[84] D. Kourtesis and I. Paraskakis, "Supporting semantically enhanced web service discovery for enterprise application integration," *Semantic enterprise application integration for business processes: Service-oriented frameworks*, IGI Global, pp.105–130, 2010.

[85] R. J. Brachman and H. J. Levesque, "Expressing Knowledge," in *Knowledge Representation and Reasoning*, Elsevier, 2004, pp. 31–47.

[86] F. Harmelen, V. Lifschitz, and B. Porter, *Handbook of Knowledge Representation*. Netherlands, Amsterdam: Elsevier, 2008.

[87] J. McCarthy, "Programs with common sense," in *Symposium on Mechanization of Thought Processes, National Physics Laboratory, Teddington, England*, November 1958.

[88] B. Nebel, "Logics for Knowledge Representation," *International Encyclopedia of the Social & Behavioral Sciences*. Elsevier, pp. 9039–9041, 2001.

[89] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[90] R. J. Brachman, "Research in natural language understanding," 1979.

[91] J. Sowa, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. San Mateo: Morgan Kaufmann, 1991.

[92] M. Minsky, "A Framework for Representing Knowledge," MIT AI Laboratory, 1974.

[93] F. Baader, "What's new in Description Logics," *Informatik-Spektrum*, vol. 34, no. 5, pp. 434–442, Apr. 2011.

[94] N. Guarino, "Formal ontology and information systems," in *Proceedings of FOIS*, 1998, vol. 98, no. 1998, pp. 81–97.

[95] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowl. Acquis.*, vol. 5, no. 2, pp. 199–220, Jun. 1993.

[96] D. Fensel, "Ontologies: A silver bullet for knowledge management and electronic-commerce (2000)," *Berlin: Spring-Verlag*.

[97] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American*, vol. 284, no. 5, pp. 34–43, May 2001.

[98] F. Baader, I. Horrocks, and U. Sattler, "Description Logics as Ontology Languages for the Semantic Web," in *Lecture Notes in Computer Science*, Springer Science & Business Media, 2005, pp. 228–248.

[99] D. L. Hendler, J., McGuinness, "The DARPA Agent Markup Language," *IEEE Intell. Syst.*, vol. 15, no. 6, pp. 67–73, 2000.

[100] D. Fensel, F. Van Harmelen, M. Klein, H. Akkermans, J. Broekstra, C. Fluit, J. van der Meer, H.-P. Schnurr, R. Studer, J. Hughes, and others, "On-to-knowledge: Ontology-based tools for knowledge management," in *Proceedings of the eBusiness and eWork*, 2000, pp. 18–20.

[101] I. Horrocks, F. van Harmelen, P. Patel-Schneider, T. Berners-Lee, D. Brickley, D. Connolly, M. Dean, S. Decker, D. Fensel, R. Fikes, and others, "Daml+ oil language specifications." Technical report, W3C, 2001. http://www. daml. org, 2001.

[102] D. L. McGuinness, F. Van Harmelen, and others, "OWL web ontology language overview," *W3C Recomm.*, vol. 10, no. 10, p. 2004, 2004.

[103] M. Horridge and P. F. Patel-Schneider, "OWL 2 web ontology language manchester syntax," *W3C Work. Gr. Note*, 2009.

[104] B. Motik, P. F. Patel-Schneider, and B. C. Grau, "OWL 2 web ontology language direct semantics," *W3C Recomm.*, vol. 27, 2009.

[105] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (XML)," *World Wide Web Consort. Recomm. REC-xml-19980210. http//www. w3. org/TR/1998/REC-xml-19980210*, vol. 16, p. 16, 1998.

[106] F. Manola, E. Miller, and B. McBride, "RDF primer," *W3C Recomm.*, vol. 10, no. 1–107, p. 6, 2004.

[107] M. Mealling and R. Denenberg, Eds., "Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations," techreport, Aug. 2002.

[108] D. Brickley and R. V Guha, "RDF vocabulary description language 1.0: RDF schema," article, 2004.

[109] D. C. Fallside and P. Walmsley, "XML schema part 0: primer second edition," *W3C Recomm.*, vol. 16, 2004.

[110] M. Kifer and H. Boley, "RIF overview," *W3C Work. Draft. W3C,(October 2009). http//www. w3. org/TR/rif-overview*, 2013.

[111] I. Horrocks and P. F. Patel-Schneider, "A proposal for an OWL rules language," in *Proceedings of the 13th conference on World Wide Web - WWW 04*, 2004.

[112] J. de Bruijn and C. Welty, "RIF, RDF and OWL compatibility," *W3C Work. Draft (July 2009). http//www. w3. org/TR/rif-rdf-owl*, 2010.

[113] E. Prud'Hommeaux, A. Seaborne, Others, E. Prud'Hommeaux, A. Seaborne, Others, E. Prud'Hommeaux, A. Seaborne, and Others, "SPARQL query language for RDF," *W3C Recomm.*, vol. 15, 2008.

[114] T. Berners-Lee, "Linked data. design issues for the world wide web," *World Wide Web Consortium. http//www. w3. org/DesignIssues/LinkedData. html*, 2006.

[115] D. Wood, Ed., *Linking Enterprise Data*. Springer Nature, 2010.

[116] B. Hu and G. Svensson, "A Case Study of Linked Enterprise Data," in *Lecture Notes in Computer Science*, Springer Science & Business Media, 2010, pp. 129–144.

[117] H.-J. Happel and S. Seedorf, "Applications of ontologies in software engineering," in *Proc. of Workshop on Sematic Web Enabled Software Engineering"(SWESE) on the ISWC*, 2006, pp. 5–9.

[118] D. Gašević, N. Kaviani, and M. Milanović, "Ontologies and Software Engineering," in *Handbook on Ontologies*, Springer Science & Business Media, 2009, pp. 593–615.

[119] M. Bergman, "The open world assumption: Elephant in the room," *AI3 Adapt. Inf.*, 2009.

[120] R. Poli, M. Healy, and A. Kameas, Eds., *Theory and Applications of Ontology: Computer Applications*. Springer Science & Business Media, 2010.

[121] D. Kourtesis and I. Paraskakis, "Governance in cloud platforms for the development and deployment of enterprise applications," *IEEE CloudCom*, 2011.

[122] D. Kourtesis and I. Paraskakis, "A registry and repository system supporting cloud application platform governance," in *Lecture Notes in Computer Science*, 2012, vol. 7221 LNCS, pp. 255–256.

[123] P. Hitzler and B. Parsia, "Ontologies and Rules," in *Handbook on Ontologies*, Springer Science & Business Media, 2009, pp. 111–132.

[124] M. Krötzsch, F. Maier, A. Krisnadhi, and P. Hitzler, "A better uncle for OWL," in *Proceedings of the 20th international conference on World wide web - WWW 11*, 2011.

[125] B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. Patel-Schneider, and U. Sattler, "OWL 2: The next step for OWL," *Web Semant. Sci. Serv. Agents World Wide Web*, vol. 6, no. 4, pp. 309–322, Nov. 2008.

[126] L. Kagal, T. Finin, and A. Joshi, "A policy language for a pervasive computing environment," in *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, 2003, pp. 63–74.

[127] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical OWL-DL reasoner," *Web Semant. Sci. Serv. Agents World Wide Web*, vol. 5, no. 2, pp. 51–53, Jun. 2007.

[128] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang, "HermiT: An OWL 2 Reasoner," *J Autom Reason.*, vol. 53, no. 3, pp. 245–269, May 2014.

[129] K. Fisler, S. Krishnamurthi, and D. J. J. Dougherty, "Embracing policy engineering," in *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER 10*, 2010.

[130] D. Lewis, K. Feeney, K. Carey, T. Tiropanis, and S. Courtenage, "Semantic-Based Policy Engineering for Autonomic Systems," in *Lecture Notes in Computer Science*, Springer Science & Business Media, 2005, pp. 152–164.

[131] S. Ross-Talbot, S. Tabet, S. Chakravarthy, and G. Brown, "A generalized RuleML-based Declarative Policy specification language for Web Services," in *W3C Workshop on Constraints and Capabilities for Web Services*, 2004.

[132] P. A. Bonatti and D. Olmedilla, "Rule-Based Policy Representation and Reasoning for the Semantic Web," in *Reasoning Web*, Springer Science & Business Media, pp. 240–268.

[133] G. Antoniou, M. Baldoni, P. A. Bonatti, W. Nejdl, and D. Olmedilla, "Rule-based Policy Specification," in *Advances in Information Security*,

Springer Science & Business Media, pp. 169–216.

[134] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, “The Ponder Policy Specification Language,” in *Lecture Notes in Computer Science*, Springer Science & Business Media, 2001, pp. 18–38.

[135] A. Uszok, J. M. M. Bradshaw, J. Lott, M. Breedy, L. Bunch, P. Feltovich, M. Johnson, and H. Jung, “New developments in ontology-based policy management: Increasing the practicality and comprehensiveness of KAoS,” in *Policies for Distributed Systems and Networks, 2008. POLICY 2008. IEEE Workshop on*, 2008, pp. 145–152.

[136] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott, “KAoS policy and domain services: toward a description-logic approach to policy representation, deconfliction, and enforcement,” in *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, 2003, pp. 93–96.

[137] A. Uszok, J. M. M. Bradshaw, and R. Jeffers, “KAoS: A Policy and Domain Services Framework for Grid Computing and Semantic Web Services,” in *Lecture Notes in Computer Science*, Springer Science & Business Media, 2004, pp. 16–26.

[138] L. Kagal, T. Berners-Lee, D. Connolly, and D. Weitzner, “Using semantic web technologies for policy management on the web,” in *Proceedings of the national conference on Artificial Intelligence*, 2006, vol. 21, no. 2, p. 1337.

[139] W. Nejdl, D. Olmedilla, M. Winslett, and C. C. Zhang, “Ontology-Based Policy Specification and Management,” in *Lecture Notes in Computer Science*, Springer Science & Business Media, 2005, pp. 290–302.

[140] R. Gavriloaie, W. Nejdl, D. Olmedilla, K. E. Seamons, and M. Winslett, “No Registration Needed: How to Use Declarative Policies and Negotiation to Access Sensitive Resources on the Semantic Web,” in *Lecture Notes in Computer Science*, Springer Science & Business Media, 2004, pp. 342–356.

[141] V. Kolovski, B. Parsia, Y. Katz, and J. Hendler, “Representing Web Service Policies in OWL-DL,” in *The Semantic Web - ISWC 2005*, Springer Science & Business Media, 2005, pp. 461–475.

[142] V. Kolovski and B. Parsia, “WS-Policy and beyond: application of OWL defaults to Web service policies,” in *Proc. of the 2nd Int. Semantic Web Policy Workshop (SWPW’06)*, 2006.

[143] M. Obitko, “RDF Graph and Syntax. Ontologies and Semantic Web,” 2007.

[144] W. W. W. Consortium and others, “RDF 1.1 Turtle: terse RDF triple language,” 2014.

[145] M.-L. Mugnier, M.-C. Rousset, and F. Ulliana, “Ontology-mediated queries for NOSQL databases,” in *AAAI: Conference on Artificial Intelligence*, 2016.

[146] F. Michel, J. Montagnat, and C. Faron-Zucker, “A survey of RDB to RDF translation approaches and tools,” PhD thesis, I3S, 2014.

[147] T. Berners-Lee, “Relational databases and the semantic web (in design issues),” *World Wide Web Consort.*, 1998.

[148] M. Rodriguez-Muro and M. Rezk, “Efficient SPARQL-to-SQL with R2RML mappings,” *Web Semant. Sci. Serv. Agents World Wide Web*, vol. 33, pp. 141–169, Aug. 2015.

[149] R. Cyganiak, “Accessing relational databases as virtual RDF graphs (2012).” Accessed, 2014.

[150] R. Greenwald, R. Stackowiak, and J. Stern, *Oracle essentials: Oracle database 12c*. “ O’Reilly Media, Inc.,” 2013.

[151] O. Erling and I. Mikhailov, “RDF Support in the Virtuoso DBMS,” in *Studies in Computational Intelligence*, Springer Science & Business Media, 2009, pp. 7–24.

[152] E. Kharlamov, E. Jiménez-Ruiz, C. Pinkel, M. Rezk, M. G. Skjæveland, A. Soylu, G. Xiao, D. Zheleznyakov, M. Giese, I. Horrocks, and others, “Optique: Ontology-Based Data Access Platform.,” in *International Semantic Web Conference (Posters & Demos)*, 2015.

[153] M. Rodriguez-Muro and D. Calvanese, “Quest, a system for ontology based data access,” in *OWL: Experiences and Directions Workshop (OWLED), Heraklion*, 2012.

[154] D. Calvanese, B. Cogrel, S. Komla-Ebri, R. Kontchakov, D. Lanti, M. Rezk, M. Rodriguez-Muro, and G. Xiao, “Ontop: Answering SPARQL queries over relational databases,” *Semant. Web*, no. Preprint, pp. 1–17, 2016.

[155] D. Calvanese, M. Giese, D. Hovland, and M. Rezk, “Ontology-Based Integration of Cross-Linked Datasets,” in *The Semantic Web - ISWC 2015*, Springer Science & Business Media, 2015, pp. 199–216.

[156] R. Kontchakov, C. Lutz, D. Toman, F. Wolter, and M. Zakharyaschev, “The combined approach to ontology-based data access,” 2011.

[157] P. Haase, T. Mathäß, M. Schmidt, A. Eberhart, and U. Walther, “Semantic Technologies for Enterprise Cloud Management,” in *Lecture Notes in Computer Science*, Springer Science & Business Media, 2010, pp. 98–113.

[158] M. Feridun and A. Tanner, “Using linked data for systems management,” in *2010 IEEE Network Operations and Management Symposium - NOMS 2010*, 2010.

[159] K. P. Joshi, “DC Proposal: Automation of Service Lifecycle on the Cloud by Using Semantic Technologies,” in *The Semantic Web - ISWC 2011*, Springer Science & Business Media, 2011, pp. 285–292.

[160] M. O'Connor and A. Das, "SQWRL: a query language for OWL," in *Proceedings of the 6th International Conference on OWL: Experiences and Directions-Volume 529*, 2009, pp. 208–215.

[161] N. Matentzoglu, J. Leo, V. Hudhra, U. Sattler, and B. Parsia, "A survey of current, stand-alone owl reasoners," in *Informal Proceedings of the 4th International Workshop on OWL Reasoner Evaluation*, 2015, vol. 1387.

[162] D. Brachman, Ronald J and Nardi, "An Introduction to Description Logics," in *The Description Logic Handbook*, Cambridge University Press, 2003.

[163] J. Tao, E. Sirin, J. Bao, and D. L. McGuinness, "Integrity Constraints in OWL.," in *AAAI*, 2010.

[164] K. Miksa, P. Sabina, and M. Kasztelnik, "Combining Ontologies with Domain Specific Languages: A Case Study from Network Configuration Software," in *Reasoning Web. Semantic Technologies for Software Engineering*, Springer Science & Business Media, 2010, pp. 99–118.

[165] S. Grimm and B. Motik, "Closed World Reasoning in the Semantic Web through Epistemic Operators.," in *OWLED*, 2005.

[166] C. Strasser and G. A. Antonelli, "Non-monotonic logic," *The Stanford Encyclopedia of Philosophy*. Stanford University, 2014.

[167] P. F. Patel-Schneider and I. Horrocks, "A comparison of two modelling paradigms in the Semantic Web," *Web Semant. Sci. Serv. Agents World Wide Web*, vol. 5, no. 4, pp. 240–250, 2007.

[168] A. A. Krisnadhi, K. Sengupta, and P. Hitzler, "Local Closed World Semantics: Grounded Circumscription for Description Logics," in *Web Reasoning and Rule Systems*, Springer Science & Business Media, 2011, pp. 263–268.

[169] L. Tao, J, Ding, L, McGuinness, D, "Instance Data Evaluation for Semantic Web-Based Knowledge Management Systems," in *42nd Hawaii International Conference on System Sciences*, 2009, pp. 1–9.

[170] B. Motik, I. Horrocks, and U. Sattler, "Adding Integrity Constraints to OWL.," in *OWLED*, 2007, vol. 258.

[171] E. Sirin, "Data Validation with OWL Integrity Constraints," in *Web Reasoning and Rule Systems*, Springer Science & Business Media, 2010, pp. 18–22.

[172] T. Bosch and K. Eckert, "Requirements on RDF constraint formulation and validation," in *Proceedings of the 2014 International Conference on Dublin Core and Metadata Applications*, 2014, pp. 95–108.

[173] H. Knublauch, J. A. Hendler, and K. Idehen, "SPIN-SPARQL inferencing notation." 2009.

[174] F. Rieckhof, H. Dibowski, and K. Kabitzsch, "Formal validation techniques for Ontology-based Device Descriptions," in *ETFA 2011*, 2011.

[175] S. Zivkovic, K. Miksa, and H. Kühn, “A Modelling Method for Consistent Physical Devices Management: An ADOxx Case Study,” in *Lecture Notes in Business Information Processing*, Springer Science & Business Media, 2011, pp. 104–118.

[176] R. Evans, D. S., Hagiu, A., & Schmalensee, *Invisible Engines: How Software Platforms Drive Innovation and Transform Industries*. MIT Press, 2008.

[177] D. C. Chou, “Rise of the cloud ecosystems,” *MSDN Blogs*, vol. 16, 2011.

[178] C. WSO2, “Governance registry brings integrity to SaaS platform,” 2012. .

[179] R. Yin, *Case study research: Design and methods*. Sage publications, 2003.

[180] S. Demeyer, “Research methods in computer science,” in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011.

[181] P. Runeson, M. Höst, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering*. Wiley-Blackwell, 2012.

[182] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet, “Architecture-level modifiability analysis (ALMA),” *J. Syst. Softw.*, vol. 69, no. 1–2, pp. 129–147, Jan. 2004.

[183] C. Pedrinaci, J. Cardoso, and T. Leidig, “Linked USDL: A Vocabulary for Web-Scale Service Trading,” in *Lecture Notes in Computer Science*, Springer Science & Business Media, 2014, pp. 68–82.

11 Appendix

```
package org.seerc.cast.regrep.lifecycle;

[...]
public class SolutionLCM extends Aspect
{
    private static final Log log = LogFactory.getLog(SolutionLCM.class);

[...]
    private Evaluation isPromotable(Resource collection, Registry registry) throws
RegistryException
    {
        if (collection == null)
        {
            log.error("collection is null");
            throw new RegistryException("collection is null");
        }

        Evaluation promotabilityEvaluation = new Evaluation();

        String currentState = collection.getProperty(currentStateProperty);

        // this shouldn't happen
        if (currentState == null)
        {
            promotabilityEvaluation.setFailureExplanation
("Critical error: No specified current state");
            log.error("Critical error: No specified current state");
            return promotabilityEvaluation;
        }

        int currentStateIndex = lifecycleStates.indexOf(currentState);

        // first check if the conditions relevant for this state are OK
        Evaluation conditionsEvaluation = checkTransitionConditions(registry,
collection, currentStateIndex);

        // null failure explanation means no failure, i.e. positive evaluation
        if (conditionsEvaluation.getFailureExplanation() == null)
        {
            // then check the maturity of all dependencies is OK relative to this state
            Evaluation maturityEvaluation = checkMaturityOfDependencies(registry,
collection, currentStateIndex);

            // if all is well, return an evaluation with null failureExplanation
            if (maturityEvaluation.getFailureExplanation() == null)
            {
                return promotabilityEvaluation;
            }
            else
            {
                return maturityEvaluation;
            }
        }
        else
            return conditionsEvaluation;
    }

    private Evaluation checkTransitionConditions(Registry registry, Resource collection,
int currentStateIndex)
    {
        Evaluation eval = null;

        switch (currentStateIndex)
        {
            case 0: // development->testing promotion
            {
                return checkDevelopmentToTestingConditions(registry, collection);
            }
            case 1: // testing->review promotion
            {
                // first run the checks of the previous transition
```

```
                eval = checkDevelopmentToTestingConditions(registry, collection);

                // if something is wrong stop, else run additional checks
                if (eval.getFailureExplanation() != null)
                    return eval;
                else
                    return checkTestingToReviewConditions(registry, collection);
            }

[...]

    private Evaluation checkTestingToReviewConditions(Registry registry, Resource
collection)
    {
        Evaluation evaluation = new Evaluation();

        // There exists a non-empty description
        // There exists a valid pricing specification file
        // There exists a valid license file
        // There exists a valid provider details file

        // check that the description is non-empty
        if (collection.getDescription() == null
                || collection.getDescription().isEmpty())
        {
            evaluation.setFailureExplanation("The description of the solution is
empty.");
            return evaluation;
        }

        // check if artefact exists in the collection root
        Resource artefact = null;
        try
        {
            artefact = registry.get(collection.getPath()
                    + RegistryConstants.PATH SEPARATOR + "pricing.xml");
        } catch (RegistryException e)
        {
            evaluation.setFailureExplanation("A pricing.xml file was not found in the
collection root.");
            return evaluation;
        }

        // check if artefact has a validation status
        String resourceValidationStatus =
artefact.getProperty(resourceValidationStatusProperty);
        if (resourceValidationStatus == null)
        {
            evaluation.setFailureExplanation("The pricing.xml file has not been validated
yet!");
            return evaluation;
        }

        // check if artefact is marked as valid
        if (!resourceValidationStatus.equalsIgnoreCase("true"))
        {
            evaluation.setFailureExplanation("The pricing.xml file is marked as
invalid.");
            return evaluation;
        }

        // check if artefact exists in the collection root
        artefact = null;
        try
        {
            artefact = registry.get(collection.getPath()
                    + RegistryConstants.PATH SEPARATOR + "license.txt");
        }
        catch (RegistryException e)
        {
            evaluation.setFailureExplanation("A license.txt file was not found in the
collection root.");
            return evaluation;
        }

        // check if artefact exists in the collection root
```

```
        artefact = null;
        try
        {
            artefact = registry.get(collection.getPath()
                    + RegistryConstants.PATH_SEPARATOR + "provider.xml");
        } catch (RegistryException e)
        {
            evaluation.setFailureExplanation("A provider.xml file was not found in the
collection root.");
            return evaluation;
        }

        // check if artefact has a validation status
        resourceValidationStatus =
artefact.getProperty(resourceValidationStatusProperty);
        if (resourceValidationStatus == null)
        {
            evaluation.setFailureExplanation("The provider.xml file has not been
validated yet!");
            return evaluation;
        }

        // check if artefact is marked as valid
        if (!resourceValidationStatus.equalsIgnoreCase("true"))
        {
            evaluation.setFailureExplanation("The provider.xml file is marked as
invalid.");
            return evaluation;
        }

        return evaluation;
    }

[...]
```

Table 55. Excerpts from the implementation of SolutionLCM.java in CAST Registry & Repository system