

AN ONTOLOGY BASED FRAMEWORK FOR CREATING PURPOSEFUL  
ONLINE COMMUNITIES

by

Murat Seyhan

B.S., Computer Engineering, Boğaziçi University, 2012

Submitted to the Institute for Graduate Studies in  
Science and Engineering in partial fulfillment of  
the requirements for the degree of  
Master of Science

Graduate Program in Computer Engineering  
Boğaziçi University

2016

AN ONTOLOGY BASED FRAMEWORK FOR CREATING PURPOSEFUL  
ONLINE COMMUNITIES

APPROVED BY:

Suzan Üsküdarlı, Ph.D. ....  
(Thesis Supervisor)

Assist. Prof. Arzucan Özgür .....

Assist. Prof. Reyhan Aydoğan .....

DATE OF APPROVAL: 15.01.2016

## ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my long-time advisor, Suzan Üsküdarlı. This work would have never been possible without her guidance and understanding. I would also like to thank my parents and fiancée for their patience and support.



## ABSTRACT

### AN ONTOLOGY BASED FRAMEWORK FOR CREATING PURPOSEFUL ONLINE COMMUNITIES

A purposeful community is a group of individuals whose actions help the community reach a set of goals. Such communities often use social network applications to communicate, coordinate, and track their activities. Twitter and Facebook are widely used for such purposes. These applications provide generic support for communication, whereas every community has different information and processing needs. For example, a community who is responding to a natural disaster will be concerned about the services and goods that need to reach victims or a community who is interested in animal rights will be interested in documenting various animals and making health services available. Clearly, the type of information for these communities is very different. Providing support for domain specific information and its processing usually involves custom applications by those who have the application building skills. Yet, many kinds of information could easily be defined by end users. If only, the means for specifying such needs were available. We propose an ontology-driven model for end users to create community-specific web applications that consume and generate Linked Data. Our model is based on the Purposeful Online Communities (POC) Core ontology, which models online communities in terms of community-specific information and activity structures. We demonstrate the use of this ontology with example communities.

## ÖZET

# AMAÇLI SANAL TOPLULUKLAR ÜRETEN ONTOLOJİ TABANLI BİR YAPI

Amaçlı topluluklar, belirli hedeflere ulaşmak için müşterek çalışan insan gruplarıdır. Bu topluluklar iletişim, koordinasyon ve durum takibi için genellikle sosyal ağ uygulamaları kullanırlar. Twitter ve Facebook bu tarz amaçlarla yoğun olarak kullanılmaktadır. Bu uygulamalar genel iletişim ihtiyaçlarını karşılar, fakat her topluluğun farklı bilgi ve bilgi işleme ihtiyaçları vardır. Örneğin, bir doğal afet müdahale topluluğu, felaketzedelere ulaştırılması gereken hizmet ve eşyalara ilgilenirken, hayvan haklarıyla ilgilenen başka bir topluluk, hayvanların kayıt altına alınması ve sağlık hizmetlerinin sağlanmasıyla ilgilenecektir. Bu toplulukların bilgi ihtiyaçları açık bir şekilde çok farklıdır. Belirli alanlara özel bilgi ve bilgi işleme ihtiyaçlarını gidermek için genellikle ihtiyaçlara göre şekillendirilmiş özel uygulamalara ihtiyaç duyulur. Bu tarz uygulamalar geliştirilmek yazılım becerileri isteyen bir iştir. Oysa, bu tarz ihtiyaçları belirtmek için kolay yollar sağlansaydı, birçok bilgi tipi son kullanıcılar tarafından tarif edilebilirdi. Bu çalışmada, son kullanıcıların amaçlı topluluklara yönelik Web uygulamaları üretmelerini sağlayan, ontoloji tabanlı bir yapı sunuyoruz. Bu yapı, sanal toplulukları özgün bilgi ve eylem çeşitleri ile modelleyen Purposeful Online Communities (POC) ontolojisini temel alır. Bu ontolojinin kullanımını örnek topluluklarla gösterilmiştir.

## TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . . .	iii
ABSTRACT . . . . .	iv
ÖZET . . . . .	v
LIST OF FIGURES . . . . .	viii
LIST OF TABLES . . . . .	xv
LIST OF ACRONYMS/ABBREVIATIONS . . . . .	xvi
1. INTRODUCTION . . . . .	1
2. BACKGROUND . . . . .	4
2.1. Representational State Transfer . . . . .	5
2.2. Semantic Web Technologies . . . . .	6
2.3. Followed Narrative Structure . . . . .	9
3. A MODEL FOR CREATING PURPOSEFUL ONLINE COMMUNITIES . . . . .	13
3.1. POC Core Ontology . . . . .	14
3.1.1. Representation of the Community Members and Their Roles . . . . .	15
3.1.2. Representation of the Community Datatypes . . . . .	15
3.1.3. Representation of the Community-Specific Processes . . . . .	21
3.1.4. Representation of Instances . . . . .	25
3.1.5. Representation of Constraints . . . . .	30
3.2. POC LDP Ontology . . . . .	30
3.3. Specification of POC Applications . . . . .	32
3.4. Execution of POC Applications . . . . .	40
3.4.1. Initialization of a POC Server . . . . .	42
3.4.2. Retrieval of a POC Server’s Representation . . . . .	43
3.4.3. Retrieval of the Community Datatypes . . . . .	44
3.4.4. Retrieval and Invocation of the Community Workflows . . . . .	46
3.4.5. Retrieval and Performance of Step Instances . . . . .	51
3.4.6. Retrieval of the Data Resources . . . . .	56
3.4.7. Execution Flow of Step Instances . . . . .	59
3.4.8. Expressions and View Templates . . . . .	60

4. EVALUATION . . . . .	62
4.1. Utilization of POC Model for a Real World Community . . . . .	62
4.1.1. A POC Specification for T3 . . . . .	63
4.1.2. Execution of a T3 POC Application . . . . .	69
4.1.2.1. Verification Attempt in the Absence of Photographs . . . . .	71
4.1.2.2. Photograph Submission . . . . .	73
4.1.2.3. Verification of Ballot Reports . . . . .	79
4.1.2.4. Accessing Verified Reports . . . . .	80
4.1.3. An Assessment of T3 POC Application . . . . .	84
4.2. An Application Generating Prototype . . . . .	85
5. RELATED WORK . . . . .	88
5.1. Related Work Utilizing Semantic Web Technologies in the Context of Online Communities . . . . .	88
5.2. Related Work on the Automated Generation of Application Behavior . . . . .	90
6. DISCUSSION AND FUTURE WORK . . . . .	93
7. CONCLUSION . . . . .	95
REFERENCES . . . . .	96

## LIST OF FIGURES

Figure 2.1.	Visualization of an example RDF graph. . . . .	7
Figure 2.2.	An RDF graph serialized in Turtle. . . . .	8
Figure 2.3.	A fragment of the Linked Datasets as of August 2014. . . . .	10
Figure 2.4.	A representation of “A <i>Car</i> is a <i>Vehicle</i> .” statement in Turtle. . .	11
Figure 2.5.	A representation of “ <i>MyCar</i> is a <i>Car</i> .” statement in Turtle. . . . .	11
Figure 2.6.	A representation of “ <i>MyCar</i> has a <i>productionDate</i> of 1965.” statement in Turtle. . . . .	12
Figure 2.7.	The declarations of all namespace prefixes that are referred, serialized in Turtle. . . . .	12
Figure 3.1.	An overview of POC Model. . . . .	14
Figure 3.2.	A visualization of the fundamental classes, properties and assertions in POC Core Ontology. . . . .	16
Figure 3.3.	A <i>DerivedDatatype</i> based on <i>ImageType</i> represented in Turtle. . .	19
Figure 3.4.	An example <i>CompositeDatatype</i> with four <i>dataFields</i> with various types, represented in Turtle. . . . .	20
Figure 3.5.	An illustration of a <i>Step</i> with two <i>inputPorts</i> and an <i>outputPort</i> . .	22



Figure 3.6.	An illustration of a <i>Workflow</i> with two <i>Steps</i> and a <i>PortPipe</i> . . . .	22
Figure 3.7.	An illustration of a <i>Workflow</i> with a <i>Step</i> and a <i>DirectPipe</i> . . . .	23
Figure 3.8.	An illustration of a <i>Workflow</i> with a <i>Step</i> and a <i>HumanPipe</i> . . . .	23
Figure 3.9.	An illustration of a <i>Workflow</i> with two <i>Steps</i> and a <i>ControlPipe</i> . . .	24
Figure 3.10.	An illustration of a <i>Workflow</i> with three <i>Steps</i> and two <i>ConditionalPipes</i> . . . . .	26
Figure 3.11.	An example <i>List</i> with three items of different types, serialized in Turtle. . . . .	26
Figure 3.12.	An example <i>Image</i> , represented in Turtle. . . . .	26
Figure 3.13.	An example <i>CompositeDataInstance</i> with three <i>fieldValues</i> , serialized in Turtle. . . . .	29
Figure 3.14.	A visualization of the classes, properties and assertions in POC LDP Ontology. . . . .	31
Figure 3.15.	An overview of specification process for POC Applications. . . . .	32
Figure 3.16.	The <i>Roles</i> in <i>SpecBirders</i> , serialized in Turtle. . . . .	34
Figure 3.17.	<i>Observation</i> serialized in Turtle. . . . .	36
Figure 3.18.	The representation of <i>ObservationList</i> , serialized in Turtle. . . . .	37
Figure 3.19.	A representation of <i>ObservationEntry</i> . . . . .	37

Figure 3.20.	<i>ObservationEntry</i> , serialized in Turtle. . . . .	38
Figure 3.21.	A representation of <i>BirdTypeEditing</i> . . . . .	39
Figure 3.22.	A representation of <i>BirdTypeEditing</i> in <i>SpecBirders</i> , serialized in Turtle. . . . .	39
Figure 3.23.	An overview of the execution process in POC Model. . . . .	40
Figure 3.24.	An HTTP request message for dereferencing the POC Server of <i>AppBirders</i> . . . . .	43
Figure 3.25.	An HTTP response message advertising a POC Server. . . . .	44
Figure 3.26.	An HTTP request message for the retrieval of the community-specific datatypes of <i>AppBirders</i> . . . . .	45
Figure 3.27.	An HTTP response message advertising the community-specific datatypes of <i>AppBirders</i> . . . . .	45
Figure 3.28.	An HTTP request message for the retrieval of a community-specific <i>Datatype</i> . . . . .	46
Figure 3.29.	An HTTP response message advertising a representation of a custom <i>Datatype</i> . . . . .	47
Figure 3.30.	An HTTP request message for the retrieval of the available workflows in <i>AppBirders</i> . . . . .	48
Figure 3.31.	An HTTP response message for the retrieval of the available workflows in <i>AppBirders</i> . . . . .	48

Figure 3.32.	An HTTP request message for the retrieval of <i>ObservationEntry</i> . . . . .	49
Figure 3.33.	An HTTP response message advertising <i>ObservationEntry</i> . . . . .	49
Figure 3.34.	An HTTP request message invoking <i>ObservationEntry</i> . . . . .	50
Figure 3.35.	HTTP response message for the invocation of <i>ObservationEntry</i> . . . . .	50
Figure 3.36.	An example HTTP request message for retrieving the <i>StepInstances</i> available for a user of <i>AppBirders</i> . . . . .	51
Figure 3.37.	An example HTTP response message advertising the <i>StepInstances</i> available for a user of <i>AppBirders</i> . . . . .	51
Figure 3.38.	An HTTP request retrieving a representation of <i>SI<sub>0</sub></i> . . . . .	52
Figure 3.39.	An HTTP response message advertising <i>SI<sub>0</sub></i> . . . . .	53
Figure 3.40.	An example SPARQL query to fetch five bird species whose labels start with "sto". . . . .	55
Figure 3.41.	An HTTP PATCH request submitting the data for the execution of <i>SI<sub>0</sub></i> . . . . .	56
Figure 3.42.	An HTTP response message, advertising the successfully performed <i>SI<sub>0</sub></i> . . . . .	57
Figure 3.43.	An HTTP request message, retrieving the data resources available for a user of <i>AppBirders</i> . . . . .	57


Figure 3.44.	An HTTP response message, advertising the data resources available for a user of <i>AppBirders</i> . . . . .	58
Figure 3.45.	An HTTP request message, retrieving an available <i>DataInstance</i> . . . . .	58
Figure 3.46.	An HTTP response message, advertising a <i>DataInstance</i> . . . . .	59
Figure 4.1.	The <i>Roles</i> defined in <i>Spec<sub>T3</sub></i> , serialized in Turtle. . . . .	64
Figure 4.2.	Definition of <i>Photo</i> in <i>Spec<sub>T3</sub></i> , serialized in Turtle. . . . .	64
Figure 4.3.	Definition of <i>VerifierReport</i> in <i>Spec<sub>T3</sub></i> , serialized in Turtle. . . . .	65
Figure 4.4.	Definition of the <i>DataInstances</i> in <i>Spec<sub>T3</sub></i> , serialized in Turtle. . . . .	66
Figure 4.5.	A visual representation of <i>PhotoSubmission</i> . . . . .	66
Figure 4.6.	Definition of <i>PhotoSubmission</i> in <i>Spec<sub>T3</sub></i> , serialized in Turtle. . . . .	68
Figure 4.7.	A visual representation of <i>ReportVerification</i> . . . . .	70
Figure 4.8.	An HTTP request message on the POC Server of <i>App<sub>T3</sub></i> . . . . .	71
Figure 4.9.	An HTTP response message advertising a representation of the POC Server of <i>App<sub>T3</sub></i> . . . . .	72
Figure 4.10.	An HTTP request for the retrieval of the <i>Workflows</i> available for a user of <i>App<sub>T3</sub></i> . . . . .	72
Figure 4.11.	An HTTP response message advertising the <i>Workflows</i> available for a user of <i>App<sub>T3</sub></i> . . . . .	73

Figure 4.12. An HTTP request message for the invocation of <i>ReportVerification</i> by $U_0$ . . . . .	73
Figure 4.13. The HTTP response message for the invocation of <i>ReportVerification</i> by $U_0$ . . . . .	74
Figure 4.14. An HTTP request for retrieving the available <i>Workflows</i> for $U_1$ . . . . .	75
Figure 4.15. An HTTP response message advertising the available <i>Workflows</i> for $U_1$ . . . . .	75
Figure 4.16. An HTTP request for the invocation of <i>PhotoSubmission</i> by $U_1$ . . . . .	76
Figure 4.17. An HTTP request for the invocation of <i>PhotoSubmission</i> by $U_1$ . . . . .	77
Figure 4.18. An HTTP request message, submitted by $U_1$ for performing $S_0$ . . . . .	78
Figure 4.19. The HTTP response message after the successful performance of $S_0$ by $U_1$ . . . . .	78
Figure 4.20. The HTTP request by $U_0$ , invoking <i>ReportVerification</i> for the second time. . . . .	79
Figure 4.21. The HTTP response message received after a successful invocation of <i>ReportVerification</i> . . . . .	81
Figure 4.22. The HTTP request message submitted by $U_0$ for the performance of $S_5$ . . . . .	82
Figure 4.23. The HTTP response message received after a successful performance of $S_5$ by $U_0$ . . . . .	82

Figure 4.24. The HTTP request message for retrieving the <i>DataInstances</i> accessible by $U_2$ . . . . .	83
Figure 4.25. The HTTP response message advertising the <i>DataInstances</i> accessible by $U_2$ . . . . .	83
Figure 4.26. The HTTP request message retrieving a representation of <i>VerifiedReports</i> . . . . .	83
Figure 4.27. The HTTP response message advertising a representation of <i>VerifiedReports</i> . . . . .	84
Figure 4.28. A screenshot of the submission page. . . . .	86
Figure 4.29. A screenshot of the endpoint displaying page. . . . .	87
Figure 4.30. A screenshot demonstrating the consumption of an endpoint advertised by the prototype. . . . .	87

## LIST OF TABLES

Table 3.1.	Derivation properties for <i>DerivedDatatypes</i> . . . . .	18
Table 3.2.	Predefined <i>Tasks</i> . . . . .	27
Table 3.3.	The results for the SPARQL query fetching bird species and their labels. . . . .	55



## LIST OF ACRONYMS/ABBREVIATIONS

POC                      Purposeful Online Community





## 1. INTRODUCTION

A Purposeful Community is a group of individuals whose actions help the community reach a set of goals. Such communities often use social network applications to communicate, coordinate, and track their activities. These applications provide generic support for communication, whereas every community has different information and processing needs. For example, a community who is responding to a natural disaster will be concerned about the services and goods that need to reach victims or a community who is interested in animal rights will be interested in documenting various animals and making health services available. Clearly, the type of information for these communities is very different. Providing support for domain specific information and its processing usually involves custom applications by those who have the application building skills. Yet, many kinds of information could easily be defined by end users. If only, the means for specifying such needs were available.

Social media applications such as Twitter [1] and Facebook [2] are widely used in computer mediated communication for organizing and communications [3, 4]. The use of such system become highly visible during events like the Occupy movement and the Arab Spring [5] and the Haiti Earthquake [6], where social media was extensively used to inform and coordinate. While these systems are very useful for transmitting information necessary to respond and coordinate, they do not support the needs of retaining or tracking information that is important for communities. In these contexts, such work, if at all, is done by persons who fetch, process, and disseminate information. Such persons are highly motivated community builders and managers who care about the community and what they are working on. They, however, generally do not possess the required technical knowledge to build applications that handle community-specific knowledge and processes.

We refer to a group of individuals that collaborate towards a set of goals as Purposeful Community, and Purposeful Online Community (POC) as a computer mediated Purposeful Community. The goals of POCs may vary considerably, so do their

information and processing needs. A POC typically utilizes multiple independent online tools to perform various tasks to suit their needs. Synchronous and asynchronous communication tools, calendars, document repositories are common tools for such purposes. Such tools offer limited interoperability (if, any) and result in community data being stored in different places and in different formats. As a result, the community knowledge is scattered into silos and cannot be effectively combined or processed. To interpret community information such as progress and status, it is typical for associated data to be manually fetched, processed, and disseminated (again, using different applications, tools, and methods). It would be very useful for community data and information to be easily accessible with interesting results automatically computed whenever possible.

In this work, we propose an ontology-driven model called Purposeful Online Communities Model (POC Model) for enabling people who are savvy in online communities to build community-centric web applications. The proposed model is based on an OWL 2 [7, 8] ontology called POC Core Ontology that we developed to model online communities with specific purposes. This ontology facilitates the specification of community-specific information and processes with a well-defined structure. Structured specification of community applications entails a straightforward integration with the structured data available on the web, or Linked Open Data [9]. The contributions of this study can be summarized as follows.

- An ontology-driven model for the specification and execution of community-centric web applications that consume and publish Linked Open Data automatically.
- An OWL 2 ontology that provides a workflow-based conceptualization for purposeful online communities.

The remainder of this document is structured as follows. In the following section, we provide a technical background for our study and discuss the narrative structure of this document. Next, we discuss the proposed model in detail. Then, the prototype we build for the model will be discussed. Later, an evaluation of the proposed model

is represented. Lastly, we discuss our current efforts and future aspects of this study, along with a conclusion.



## 2. BACKGROUND

In this chapter, we discuss the terminology and technologies that our work depends on.

The model we propose is inspired by crowdsourcing, human computation systems, and computer mediated workflows.

Human computation is a research topic focused on exploiting human intelligence for computational tasks that are computationally difficult for computers, but easy for humans to solve [10]. Human computation systems form purposeful communities, where the actions of the community members help the community reach certain goals, even though the members may not be aware of what the goals are. Human computation applications are utilized for various objectives. reCAPTCHA [11] is a well known such application that is used to verify human users on the web. It presents the user two pieces of distorted text, which they are supposed to type in text boxes in order to gain access to a website. The system only knows the correct answer for one of the text images, the other is from an un-deciphered optical reader. Humans happen to be very good to deciphering distorted text. With this system, hundreds of thousands of man hours per day were accumulated via very small actions of a massive number of users and a great amount of digitization work was accomplished. Another example of such systems is called *games with a purpose* [12], which are games that utilize the contributions of their players for some computational tasks. Crowdsourcing marketplaces, such as Mechanical Turk (MTurk) [13] enable human workers to perform computational tasks, such as data labeling, for small monetary rewards. MTurk is reported to be a source of inexpensive and high-quality data for research [14]. Another example is Duolingo [15] that teaches languages while automatically translating the Web to all major languages.

Computer mediated workflows typically refer to the automation of whole or partial business processes [16]. They consist of activities that are interdependent in according to a set of procedural rules. Dependencies arise when activities must be ordered to

satisfy some constraints, such as need for information generated in a particular activity. The work associated with activities may be performed by humans or computers. Our model aims to apply the workflow notion to online communities that define their own workflows.

## 2.1. Representational State Transfer

Representational State Transfer (REST) [17] is an architectural style for distributed hypermedia systems. REST is an abstract model of the Web architecture. It is composed of the principles that guided the design of Uniform Resource Identifier (URI) [18] and HTTP/1.1 [19], and is intended as a guideline for development of overall Web applications [20]. REST style is composed of six architectural constraints, namely: client-server, statelessness, cache, layered system, code on demand (optional), and uniform interface. A system complying with these constraints is referred as a RESTful system [21]. Client-server architectural style is the separation of user interface concerns from the data store concerns, which is typical for web applications. This constraint aims to improve scalability by simplifying the server constraints and improve portability of the user interface. Second constraint is the statelessness of the client-server communication, meaning that the requests from client to server must contain all the information needed to process the request. Statelessness implies that the session state must be stored on the client. This constraint aims to improve visibility, reliability and scalability of the architecture, yet introduces an overhead of sending repetitive data in requests in a shared context. Cache constraint dictates that the data within a response to a request must be implicitly or explicitly labeled as cacheable or non-cacheable, where cacheable responses can be cached and reused by the client. This constraint aims to yield efficiency and scalability, but is noted to introduce a possible decrease of reliability if stale cache data is not invalidated properly. Layered system constraint requires the components of a system to form hierarchical layers, and to have knowledge of only the immediate layers they interact with. Layered system architecture limits the overall complexity of a system. As a side effect, processing of data on multiple layers introduces a latency, diminishing the performance. Code-on-demand style allows a client component that cannot process a set of resources to ask

a remote server for the code to process those resources, receive the code, and execute it locally. It is the only optional constraint of REST, therefore a system not satisfying this constraint may still be RESTful. Uniform interface constraint is a distinguishing feature of REST. It implies the decoupling of the implementations from the services they provide, to provide a standardized interface for the clients. A uniform interface is obtained if a system satisfies the following four constraints. First, any concept that can be targeted with a hypertext reference must fit within the definition of a resource, where a resource is a conceptual mapping to a set of resource identifiers and/or representations. The messages must identify resources using resource identifiers, which are typically URIs in RESTful web applications. Secondly, the resources must be manipulated through resource representations, e.g. an XML document in a RESTful web API. Third, the messages must be self-descriptive. This must be ensured by the stateless interaction between requests, the explicit representation of cacheability in responses, and the utilization of standard methods and media types to indicate semantics and exchange information. Last requirement for a uniform interface is the utilization of hypermedia as the engine of the application state. This implies that the interaction of a client with a server must depend only on the hypermedia provided by the server, and must not assume a response structure beyond that defined by the specification of the utilized media type [21].

## 2.2. Semantic Web Technologies

Traditional web technologies enable us to link related documents, or parts of document, to one another. These hyperlinks do not carry a well-defined meaning, and their interpretation is often left to human end users. Many applications we use over the web aggregate data relevant for us. Combination of such distributed data outside the context of individual applications would potentially yield powerful new applications. This requires the data to be interlinked with well-defined meaning, so that it can be discovered and processed by automated tools properly. Such data is typically not interlinked at all, and, therefore, is segregated into silos formed by each application. The World Wide Web Consortium (W3C) [22] aims to address this issue with a set

of standard technologies and principles to facilitate meaningful representation, linking and processing of distributed data elements across the Web. The Semantic Web, or the Web of Data, refers to a Web enriched with such standards, on which the data is interlinked with well-defined meaning [23].

A fundamental building block of the Semantic Web is Resource Description Framework 1.1 (RDF) [24, 25]. RDF is a model providing a standard way to identify data elements and express relations between them. In RDF, relations between two resources are expressed with an RDF triple, which is composed of a subject, predicate and an object. The structure of an RDF triple resembles an elementary sentence, e.g. “Jack speaks English.”. The subject and the object of an RDF triple is a IRI, Literal or a Blank node, and the predicate is typically a URI [18]. This facilitates the relations expressed in RDF to possess well-defined meanings, unlike hyperlinks between web documents. Data expressed in RDF forms a directed graph, where each triple identifies an edge, directed from the subject of the triple, towards its object. Figure 2.1 depicts an example RDF graph.

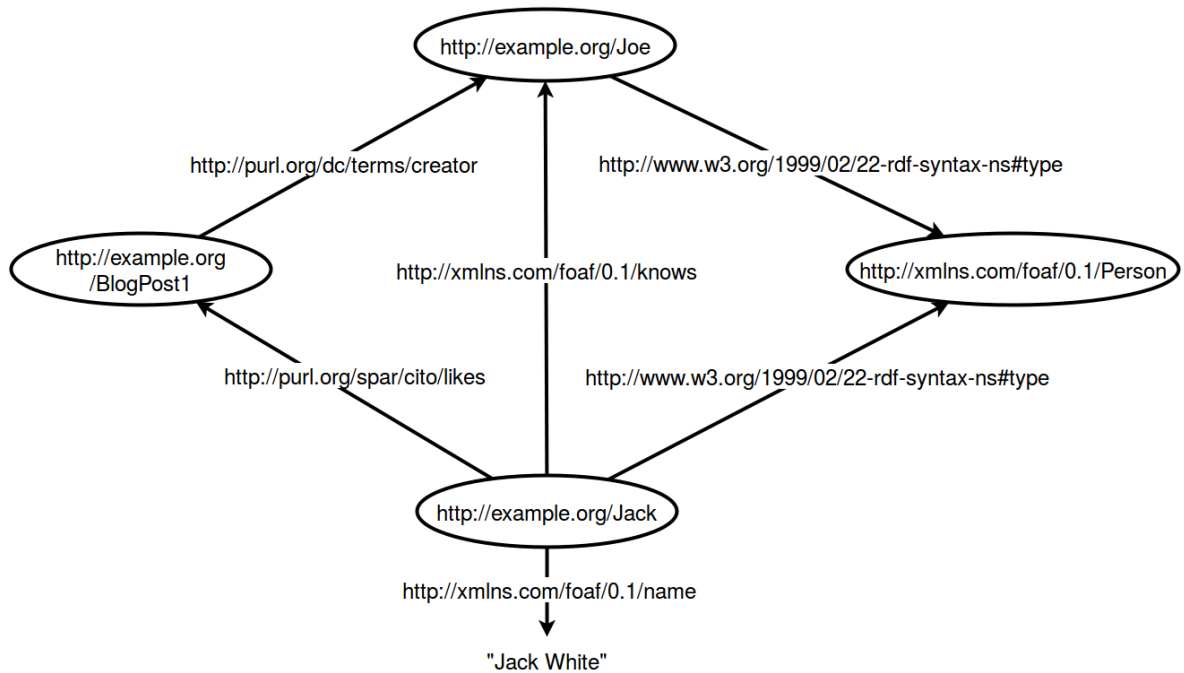


Figure 2.1. Visualization of an example RDF graph.

RDF data can be serialized in various formats, e.g. Turtle [26], JSON-LD [27] and RDF/XML [28]. Figure 2.2 gives a Turtle serialization of the RDF graph depicted in Figure 2.1.

```

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix poc: <http://www.cmpe.boun.edu.tr/soslab/ontologies/poc/core#>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix cito: <http://purl.org/spar/cito/>.
<http://example.org/Joe> rdf:type foaf:Person.
<http://example.org/BlogPost1> dcterms:creator
    <http://example.org/Joe>.
<http://example.org/Jack> rdf:type foaf:Person;
    foaf:knows <http://example.org/Joe>;
    foaf:name "Jack White";
    cito:likes <http://example.org/BlogPost1>.

```

Figure 2.2. An RDF graph serialized in Turtle.

Other core technologies by W3C enable us to query, model, infer and advertise RDF data effectively. SPARQL 1.1 [29] is query language for RDF data, with a syntax resembling SQL. RDF Schema 1.1 (RDFS) [30] is a data-modelling vocabulary for RDF data. OWL 2 Web Ontology Language (OWL 2) [7, 8] is a knowledge representation language, enabling specification of domain knowledge in terms of concepts and their relationships. Resulting ontologies are explicit specifications that may be published on the Web, thereby provides means for shared semantics across applications. Rule languages such as Semantic Web Rule Language (SWRL) [31] provides more expressive power on top of OWL 2, enabling conceptualization of more complex domains. Structured representation of the domain knowledge with the mentioned technologies facilitate inference of new relations from the existing data.



In order for the Semantic Web to become a reality, a substantial amount of data need to be published and interlinked in its standard formats. Such data is referred as Linked Data [9], or Linked Open Data if it is publicly accessible. Figure 2.3 depicts a graph of significant Linked Datasets as of August 2014 [32]. A notable effort towards such Linked Data is DBpedia project [33,34]. DBpedia project is a community effort for extracting information from Wikipedia [35] and advertising the extracted data through the Semantic Web standards. English version of the DBpedia data set is reported to contain more than 5 million entities [36]. Upper Mapping and Binding Exchange Layer (UMBEL) [37] provides UMBEL Vocabulary, which is intended as a standard framework for domain ontologies to structure Linked Open Data. UMBEL Reference Concept Ontology is an ontology complying UMBEL Vocabulary that provides a set of generic classes to be referenced by various Linked Datasets.

Linked Data Platform 1.0 (LDP) [38,39] is a recently published specification by W3C, defining a set of rules for accessing, updating, creating and deleting of Linked Data over HTTP. An LDP Server is defined as an HTTP server [40] that conforms the rules specified by LDP to advertise data. An LDP Client is defined an HTTP client [40] that conforms the rules specified by LDP to consume data from an LDP Server. Supporting HTTP PATCH method [41] for the modification of advertised data is optional for LDP Servers. [39] defines a patch format for LDP Servers that support HTTP PATCH.

### 2.3. Followed Narrative Structure

In this section, we discuss some key properties of the narrative structure followed through this document.

This document contains a substantial amount of discussion on the structure of OWL 2 ontologies. For the sake of convenience, we refer to the entities in these ontologies, such as OWL 2 classes, their instances and properties, with an italic font, e.g. “*Car* is a class in the ontology.” Class names follow a camel case notation with a capital first letter, e.g. *CarWindow*, whereas property names follow a camel case

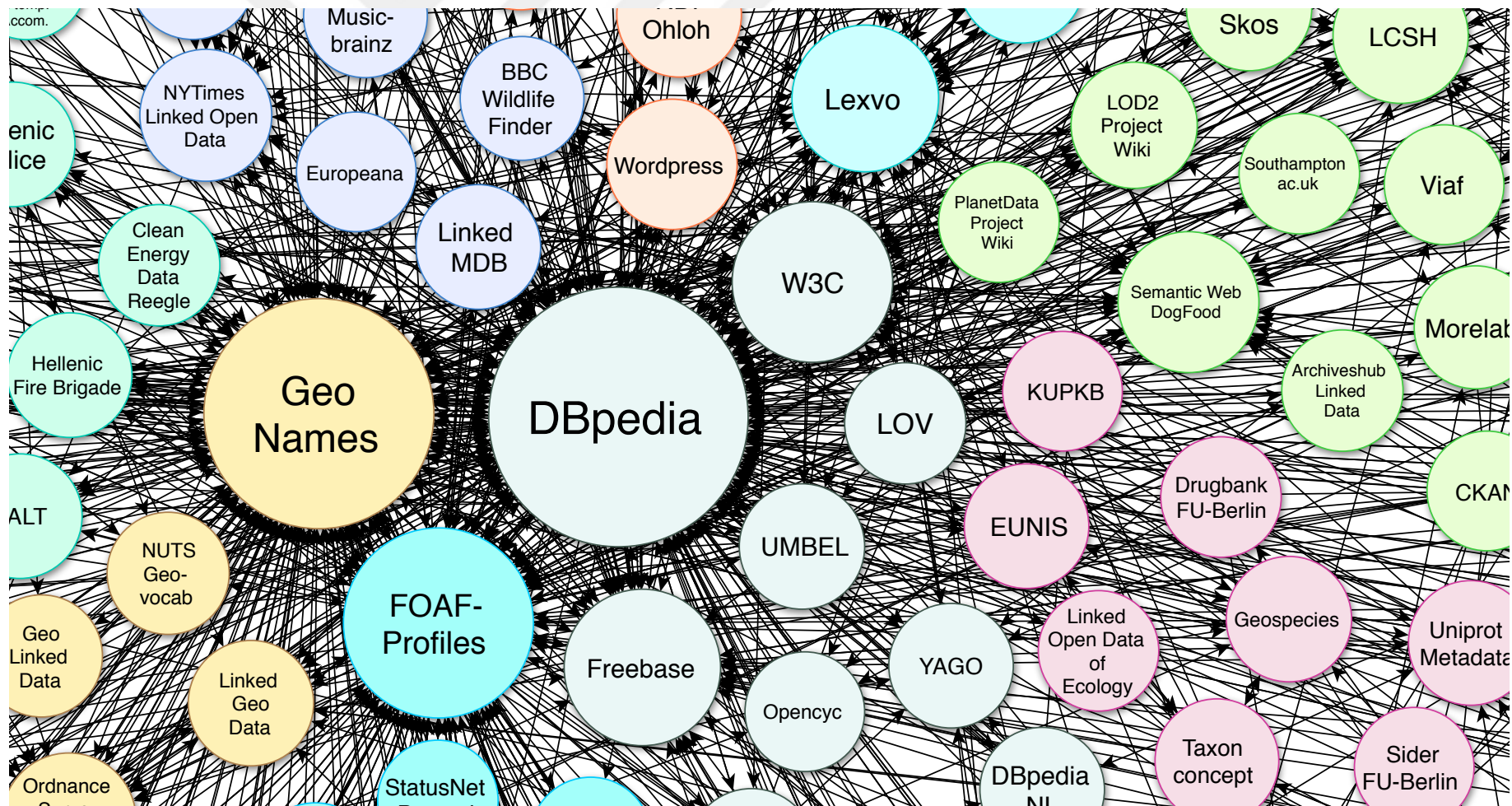


Figure 2.3. A fragment of the Linked Datasets as of August 2014.

notation with lowercase first letter, e.g. *productionDate*. The reader should also note that, when a class name is represented with a prepended Article or in plural form, the referred entities are the instances of the class, e.g. “A *Step* is used to represent a single operation.” or “*Steps* are used to represent operations.” In this context, a statement such as “A *Car* is a *Vehicle*.” indicates that the instances of *Car* class are also instances of *Vehicle* class, implying that *Car* is a subclass of *Vehicle*. A representation of such statement in RDF is depicted in Figure 2.4.

```
poc:Car rdfs:subClassOf poc:Vehicle.
```

Figure 2.4. A representation of “A *Car* is a *Vehicle*.” statement in Turtle.

A statement such as “*MyCar* is a *Car*.” indicates that *MyCar* is an actualized instance of *Car*. A representation of such statement in RDF is illustrated in Figure 2.5.

```
poc:MyCar rdf:type poc:Car.
```

Figure 2.5. A representation of “*MyCar* is a *Car*.” statement in Turtle.

Moreover, if a property name is mentioned in the context of a class instance, the actual referred entity is the value that the property links to, e.g. “*MyCar* has a *productionDate* of 1965.” A representation of such statement in RDF is depicted in Figure 2.6.

The remainder of this document contains many listings representing RDF data, which are all serialized in Turtle [26]. For the sake of convenience, the namespace prefixes referred by such listings are omitted. The declaration of all namespace prefixes referred by such listings are represented in Figure 2.7.

```
poc:MyCar poc:productionDate 1965.
```

Figure 2.6. A representation of “*MyCar* has a *productionDate* of 1965.” statement in Turtle.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix sioc: <http://rdfs.org/sioc#>.
@prefix sioc-services: <http://rdfs.org/sioc/services#>.
@prefix gn: <http://www.geonames.org/ontology#>.
@prefix ma: <http://www.w3.org/ns/ma-ont#> .
@prefix umbel-rc: <http://umbel.org/umbel/rc/>.
@prefix poc: <http://www.cmpe.boun.edu.tr/soslab/ontologies/poc/core#>.
```

Figure 2.7. The declarations of all namespace prefixes that are referred, serialized in Turtle.

### 3. A MODEL FOR CREATING PURPOSEFUL ONLINE COMMUNITIES

We propose a model for POCs by supporting the definition of community-specific information and processes. Community-specific information is defined as user defined data types by specifying a collection of typed fields. For example, community of bird lovers may be interested in birds that are pets. Furthermore, they may want to collect information about these pets in terms of the picture of the bird (as an image), the type of the bird (name of species), and the location of the bird (as a geo-location). Community processes are collections of simple tasks to create, access, and process community-specific information. Tasks are coordinated via workflows that specify the control and data flow between the tasks.

Purposeful communities tend to be self organizing and initiated by a few dedicated core members. We refer to such persons as Community Builders and expect that they would perform the lion share of defining the community information and processes. Most of the community members will be participants who contribute to the community work as specified by the the Community Builders.

To address the specification of community-specific information and processes, we introduce an ontology-driven model called Purposeful Online Communities Model (POC Model). Figure 3.1 shows an overview of the model. There are two major concerns addressed by POC Model:

- the *specification* of community-specific information and workflows
- the *execution* of the application that realizes a specification.

The specification phase yields a POC Specification that specifies the community-application. A POC Specification, in turn, is used to drive the execution of the corresponding POC Application. POC Clients are clients that interact with POC Applica-

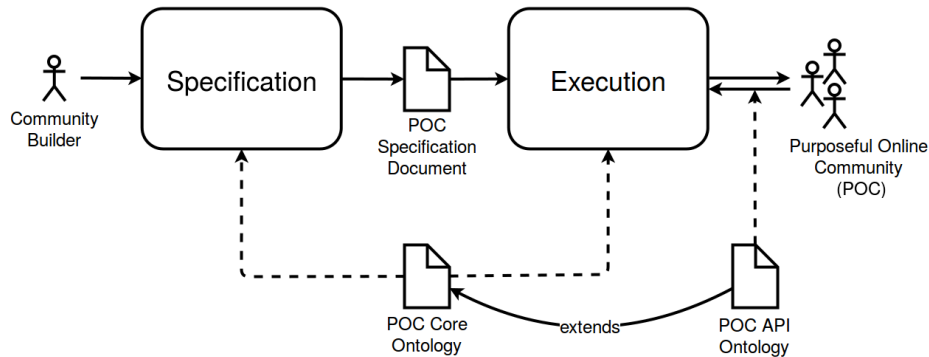


Figure 3.1. An overview of POC Model.

tions. In other words, to realize the specification. The POC LDP Ontology ontology extends POC Core Ontology with a vocabulary to exchange data between POC Applications and POC Clients.

We introduce, POC Core Ontology, which is an ontology to specify Purposeful Online Community communities (POC Specifications). POC Core Ontology is specified with OWL 2 [7, 8]. POC Applications provide endpoints that advertise their data according to a given POC Specification. Data is advertised consistent with Linked Data standards [38]. POC Clients consume data served by POC Applications.

The remainder of this chapter focuses on the specification of a Purposeful Online Community and the corresponding POC Core Ontology concepts and relations.

### 3.1. POC Core Ontology

POC Core Ontology is an OWL 2 ontology that specifies a conceptual model for POCs. POC Core Ontology is identified by the namespace URI `http://www.cmpe.boun.edu.tr/soslab/ontologies/poc/core#`. The namespace prefix `poc` is used throughout this thesis to refer to this namespace.

The design of the POC Core Ontology adheres to the recommendations and standard practices provided by W3C. Existing ontologies (or vocabularies) have been

utilized in POC Core Ontology. Dublin Core Vocabulary (DCTERMS) [42, 43] is used for representing generic meta-data, Semantically-Interlinked Online Communities (SIOC) [44, 45] for concepts related to online communities, and Ontology for Media Resources [46] for media resources. Also, XSD datatypes [47] is utilized with the datatype derivation mechanism supported by OWL 2.

The fundamental classes in POC Core Ontology and their relations are shown in Figure 3.2. In this figure, circular nodes represent classes. Classes that are external to POC Core Ontology are labeled as “external”. Edges are used to represent the domain and range of properties. Dotted edges represent assertions. The *POCConcept* is the root class in the ontology, thus, all other classes in the `poc` namespace is a subclass of *POCConcept*.

### 3.1.1. Representation of the Community Members and Their Roles

Members of a POC Application community are represented with *UserAccounts*. All the activities of a community member through the application are associated with their account. In other words, it represents their identity.

Members of a POC may possess various authorities according to the the practices of a community. *Roles* define such authorities, which can be assigned to members. A user may have zero or more *Roles* granted by the community. This assignment of a *Role* to a *UserAccount* is represented with the *has\_function* property of SIOC (*sioc:has\_function*).

### 3.1.2. Representation of the Community Datatypes

Community data types consist of literal, primitive, and community-defined data types. XSD datatypes [47] represent literal types, such as integer, string, and date. This approach is common practice when specifying OWL 2 ontologies. Community specific datatypes are represented with *Datatypes*, which has two subclasses: *PrimitiveDatatype* and *DerivedDatatype*.

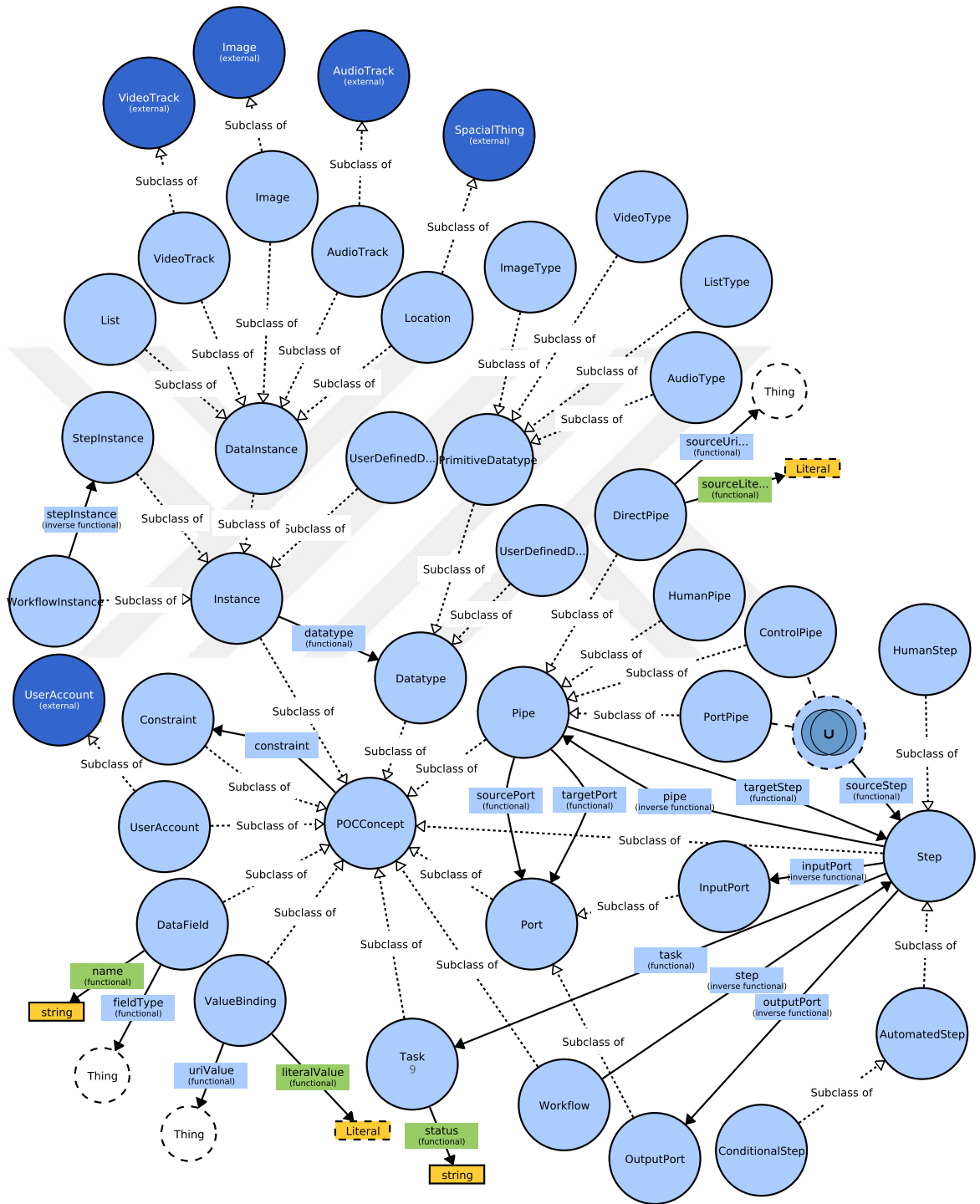


Figure 3.2. A visualization of the fundamental classes, properties and assertions in POC Core Ontology.



*PrimitiveDatatypes* represent the basic, non-literal datatypes that are commonly manipulated by POCs. *ListType* is a *PrimitiveDatatype* for lists. *MediaType* is a subclass of *PrimitiveDatatype*, representing the types for the Audiovisual media resources. *ImageType*, *AudioTrack* and *VideoTrack* are *PrimitiveDatatypes* representing types for images, audiotracks and videotracks.

OWL 2 [8] supports a mechanism to derive new restricted datatypes from the existing literals, e.g. a string with a maximum length of 144 characters. This, however, does not give us enough expressivity to define all custom types that would be needed by individual POCs. *UserDefinedDatatypes* represent community-specific types for the data resources utilized within a POC. *UserDefinedDatatype* class has two subclasses; namely, *DerivedDatatype* and *CompositeDatatype*.

*DerivedDatatypes* are constrained derivations from other *Datatypes*, e.g. an *ImageType* for images that are not allowed to have a frame height larger than 800 pixels. A *DerivedDatatype* has one or more *baseDatatypes*, which are the *Datatypes* that the database derivation is based on. The *baseDatatype* of a *DerivedDatatype* can be a *PrimitiveDatatype* or another *DerivedDatatype*. The properties that can be used to define restrictions on *DerivedDatatypes* are represented in Table 3.1.

An example *DerivedDatatype* representation is depicted in Figure 3.3. It defines a *DerivedDatatype* based on *ImageType*, which restricts its instances to have a maximum frame width and height of 200 pixels, and a minimum file size of 100 kilobytes.

*CompositeDatatypes* are *UserDefinedDatatypes* representing types for data resources consisting of one or more named fields of various types. A *CompositeDatatype* has one or more *dataFields*, which are *DataFields*. A *DataField* has a string *label*, a *fieldType*, and zero or one *sourceService*. It also has an optional boolean flag, *isRequired*. If *isRequired* flag of a *CompositeDatatype* is true, its instances must contain a value corresponding to the *DataField*. If *isRequired* flag is false or missing, the values for the *DataField* can be left blank. A *DataField* is uniquely identified with its *label* among the *dataFields* of a *CompositeDatatype*. *fieldType* of a *DataField* specifies what

Table 3.1. Derivation properties for *DerivedDatatypes*.

<b>Name</b>	<b>Range</b>	<b>Description</b>
<i>maxFrameWidth</i>	xsd:integer	The maximum frame width in pixels for <i>DerivedDatatypes</i> based on <i>ImageType</i> and <i>AudioType</i> .
<i>minFrameWidth</i>	xsd:integer	The minimum frame width in pixels for <i>DerivedDatatypes</i> based on <i>ImageType</i> and <i>AudioType</i> .
<i>maxFrameHeight</i>	xsd:integer	The maximum frame height in pixels for <i>DerivedDatatypes</i> based on <i>ImageType</i> and <i>AudioType</i> .
<i>minFrameHeight</i>	xsd:integer	The minimum frame height in pixels for <i>DerivedDatatypes</i> based on <i>ImageType</i> and <i>AudioType</i> .
<i>maxTrackLength</i>	xsd:integer	The maximum file size in kilobytes for <i>DerivedDatatypes</i> based on <i>MediaTypes</i> .
<i>maxTrackLength</i>	xsd:integer	The maximum file size in kilobytes for <i>DerivedDatatypes</i> based on <i>MediaTypes</i> .
<i>maxFileSize</i>	xsd:integer	The maximum file size for <i>DerivedDatatypes</i> based on <i>MediaTypes</i> .
<i>minFileSize</i>	xsd:integer	The maximum file size for <i>DerivedDatatypes</i> based on <i>MediaTypes</i> .
<i>scaleWidth</i>	xsd:integer	The target frame width in pixels for resources of <i>DerivedDatatypes</i> based on <i>ImageType</i> to be scaled before their storage.
<i>scaleHeight</i>	xsd:integer	The target frame height in pixels for resources of <i>DerivedDatatypes</i> based on <i>ImageType</i> to be scaled before their storage.
<i>maxSize</i>	xsd:integer	The maximum size for <i>DerivedDatatypes</i> based on <i>ListType</i> .

```

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix poc: <http://www.cmpe.boun.edu.tr/soslab/ontologies/poc/core#>.
<http://example.org/SmallImageType> a poc:DerivedDatatype;
    poc:baseDatatype poc:ImageType;
    poc:maxFrameWidth 200;
    poc:maxFrameHeight 200;
    poc:minFileSize 100.

```

Figure 3.3. A *DerivedDatatype* based on *ImageType* represented in Turtle.

kind of data is allowed for the *DataField*. It is typically a XSD datatype or a *Datatype*, yet can be any other type that can be identified with a URI. Values for a *DataField* can be expected to be retrieved from an external Linked Dataset. Endpoint of such a datasets is identified by the *sourceService* of a *DataField*. Figure 3.4 demonstrates an example *CompositeDatatype*. It defines a *CompositeDatatype* with a label "Haircut", which has four *dataFields*. The first *DataField* has "Name" as its *label* and a *fieldType* of string. This *DataField* is specified to be required, indicating that it does not allow blank values. The second *DataField* has "Cost" as its *label*, and a *fieldType* of decimal. The third *DataField* has "Difficulty" as its *label*, and a *fieldType* of a datatype derived from integer, restricted to be greater than or equal to 0 and less than or equal to 10. The fourth *DataField* has "Photo" as its *label*, and a *fieldType* of the *DerivedDatatype* we defined in Figure 3.3.

In addition to the mentioned types, POC Core Ontology introduce three special literal types: *Expression*, *ViewTemplate* and *View*. These literal types are derived from string of XML Schema Definition Language [47], with the datatype derivation mechanism supported by OWL 2 Web Ontology Language [8]. POC Core Ontology does not impose any further meaning on these literal types, and their utilization in POC Model will be discussed in Section 3.4.8.

```

<http://example.org/Haircut> a poc:CompositeDatatype;
  rdfs:label "Haircut";
  poc:dataField
    [ a poc:DataField;
      rdfs:label "Name";
      dcterms:description "Name of the haircut."@en;
      poc:isRequired true;
      poc:fieldType xsd:string
    ], [ a poc:DataField;
      rdfs:label "Cost";
      dcterms:description "Recommended cost of the haircut in USD."@en;
      poc:fieldType xsd:decimal
    ], [ a poc:DataField;
      rdfs:label "Difficulty";
      dcterms:description "A rating (0-10) indicating the difficulty to
        perform the haircut."@en;
      poc:fieldType [ rdf:type rdfs:Datatype ;
        owl:onDatatype xsd:integer ;
        owl:withRestrictions ( [xsd:minInclusive 0] [xsd:maxInclusive
          10] )
      ] .
    ], [ a poc:DataField;
      rdfs:label "Photo";
      poc:fieldType <http://example.org/SmallImage>
    ].

```

Figure 3.4. An example *CompositeDatatype* with four *dataFields* with various types, represented in Turtle.

### 3.1.3. Representation of the Community-Specific Processes

POCs have diverse purposes and their own ways to reach their goals. Applications that target specific POCs should, thereby, have the capability to perform custom processes utilized by each community. Such processes often involve multiple steps, to be performed by different members of the community or by the application itself, automatically. Representation of such community-specific processes are the essence of our conceptualizations of POCs.

*Workflows* represent community-specific processes, which are pipelines composed of one or more basic operations. A *Workflow* encapsulates one or more interdependent *steps* and zero or more *pipes*.

Each *step* of a *Workflow* is a *Step*. A *Step* represents an invocation of a predefined operation within a *Workflow*, which can be executed autonomously or manually by a community member. A *Step* has a *task*, zero or one *viewTemplate*, zero or more *inputPorts* and zero or more *outputPorts*. The *inputPorts* and *outputPorts* of a *Step* are *Ports*. A *Port* is a data handle that is used to transfer data from or to a *Step*. Each *Port* of a *Step* is uniquely identified with its *label* among its *inputPorts* and *outputPorts*. The *task* of a *Step* identifies what kind of operation is involved by the *Step*. For the sake of clarity, we will discuss other components of *Workflows* before *Tasks*. The *viewTemplate* of a *Step* is a *ViewTemplate*, representing the information to be displayed to the users who enact the *Step*. Figure 3.5 depicts a *Step* identified as  $S_0$ , with two *inputPorts* and an *outputPort*. The first *inputPort* of  $S_0$  has "input0" as its *label*, and the second has "input1". The *outputPort* of  $S_0$  has "output0" as its *label*.

*Pipes* represent data and control dependencies of *Steps*. They are typically used to specify how *Steps* are related to one another in a *Workflow*. There are four types of *Pipes* to represent the nature of the dependency: *PortPipe*, *DirectPipe*, *HumanPipe* and *ControlPipe*.

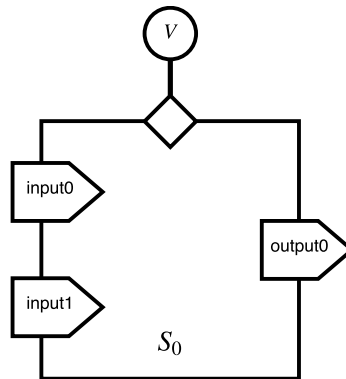


Figure 3.5. An illustration of a *Step* with two *inputPorts* and an *outputPort*. The large rectangle represents the *Step*. Small diamond and the circle labeled  $V$  represent a *viewTemplate*. Small pentagons represent *inputPorts* and *outputPorts* of the *Step*. The text on each pentagon represent the *label* of the *Port*.

*PortPipes* represent the data flow between *Steps*. A *PortPipe* is used to transmit the outputs of a *Step* to another. Each *PortPipe* has a *sourceStep*, a *sourcePort*, a *targetStep* and a *targetPort*. The *sourcePort* identifies which *outputPort* of the *sourceStep* is the source of the data binding; and, the *targetPort* identifies which *inputPort* of the *targetStep* is the target of the data binding. Figure 3.6 depicts a *Workflow* with two *Steps*;  $S_0$  and  $S_1$ , and a *PortPipe*.  $S_0$  has an *outputPort* labeled "output".  $S_1$  has an *inputPort* labeled "input". The depicted *PortPipe* has  $S_0$  as its *sourceStep*, "output" as its *sourcePort*,  $S_1$  as its *targetStep* and "input" as its *targetPort*.

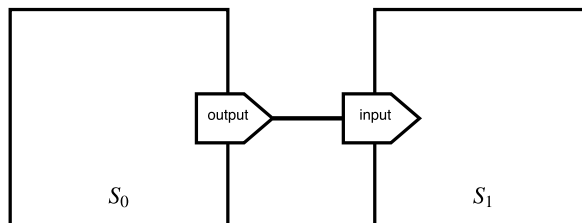


Figure 3.6. An illustration of a *Workflow* with two *Steps* and a *PortPipe*. The *PortPipe* is represented with a connecting line between the two *Ports*.

*DirectPipes* are used to insert values directly to *Steps*. Each *DirectPipe* has a *targetStep*, *targetPort*, and a *sourceValue*. *sourceValue* represents the data to be

inserted, which can be a literal value or a *DataInstance*. *targetPort* specifies which *inputPort* of *targetStep* will be bound with the data. Figure 3.7 depicts a *Workflow* with a single *Step*,  $S_0$ , and a *DirectPipe*.  $S_0$  has a single *inputPort*, labeled "input". The *DirectPipe* has  $S_0$  as its *targetStep*, "input" as its *targetPort*, and "value" as its *sourceValue*.

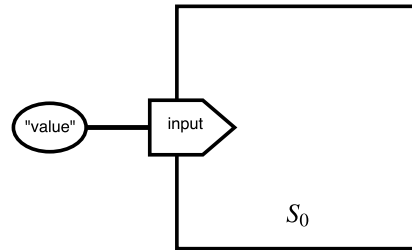


Figure 3.7. An illustration of a *Workflow* with a single *Step* and a *DirectPipe*. The *DirectPipe* is represented with an ellipse and a connecting line between the ellipse and the *Port*. The text inside the ellipse is the *sourceValue* of the *DirectPipe*.

*HumanPipes* represent manual data entries by the users to the *Steps*. A *HumanPipe* has a *targetStep*, a *targetPort*. *targetPort* of a *HumanPipe* identifies the *inputPort* of *targetStep* that the input from users is intended for. Figure 3.8 depicts a *Workflow* with a single *Step*,  $S_0$ , and a *HumanPipe*. The *HumanPipe* has  $S_0$  as its *targetStep* and "input" as its *targetPort*.

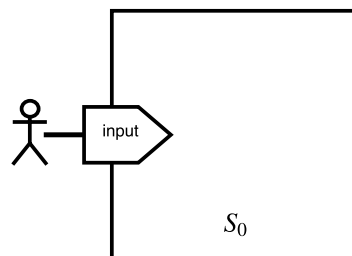


Figure 3.8. An illustration of a *Workflow* with a single *Step* and a *HumanPipe*. The *HumanPipe* is represented with a stick figure and a connecting line between the stick figure and the *Port*.

*ControlPipes* represent the control dependencies between *Steps* with no data flow. Each *ControlPipe* has a *targetStep* and a *sourceStep*, which are two distinct *Steps* of a

*Workflow*. The *targetStep* of a *ControlPipe* cannot be executed before the *sourceStep* is successfully completed. Figure 3.9 depicts a *Workflow* with two *Steps*;  $S_0$  and  $S_1$ , and a *ControlPipe*. The *ControlPipe* has  $S_0$  as its *sourceStep* and  $S_1$  as its *targetStep*.

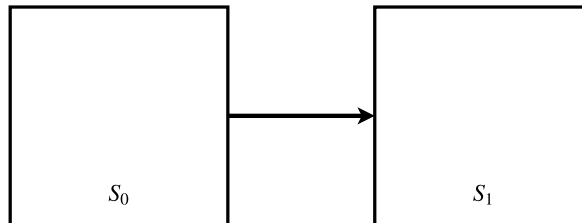


Figure 3.9. An illustration of a *Workflow* with two *Steps* and a *ControlPipe*. The *ControlPipe* is represented with an arrow between the *Steps*.

*ConditionalPipes* are *ControlPipes* that represent control dependencies between *Steps* that determined by the value of a certain output. Each *ConditionalPipe* has a *sourcePort*, which identifies which *outputPort* of its *sourceStep* determines the control flow. There are two types of *ConditionalPipes*: *TruePipe* and *FalsePipe*. A *TruePipe* represents a control flow that is only enacted if the *sourcePort* evaluates true. Similarly, a *FalsePipe* represents a control flow that is only enacted if the *sourcePort* evaluates false. Figure 3.10 depicts a *Workflow* with three *Steps*;  $S_0$ ,  $S_1$  and  $S_2$ , a *TruePipe* and a *FalsePipe*. The *TruePipe* has  $S_0$  as its *sourceStep*, "output" as its *sourcePort* and  $S_1$  as its *targetStep*. The *FalsePipe* has  $S_0$  as its *sourceStep*, "output" as its *sourcePort* and  $S_2$  as its *targetStep*.

*Tasks* are predefined simple operations to be performed within *Workflows* Any, complex or simple, process to be performed within a POC involves the enactment of one or more *Tasks*. *Tasks* are invoked by *Steps* of a *Workflow*, with the values provided for the *inputPorts* of a *Step*. Similar to the *Steps* that invoke them, each *Task* has zero or more *inputPorts* and zero or more *outputPorts*. *Tasks* resemble functions in a functional programming environment, since *Steps*, like function calls, invoke them with a set of inputs and pass its outputs to the next *Step*, or function call. However, this analogy is not perfect, as the data being passed to *Tasks* is mutable, and can be manipulated by the *Task*. We introduce a set of predefined *Tasks*, details of which are



represented in Table 3.2. Implementations of POC Model can extend this predefined set in order to facilitate more complex use cases.

### 3.1.4. Representation of Instances

Various *POCConcepts* that are discussed above can have multiple enactment, or instances, within a POC. For instance, there can be multiple resources that are instances of the same *DerivedDatatype*, or a *Workflow* can be enacted multiple times. Such instances are represented with *Instance* class in POC Core Ontology. *creator* and *created* properties of DCTERMS are exploited for identifying the *UserAccounts* that have caused the creation of *Instances* and the time of their creation. *Instance* class has three subclasses; namely *DataInstance*, *WorkflowInstance* and *StepInstance*.

Literal values, such as numbers, strings, dates, are represented through XSD datatypes [47] in POC Core Ontology. More complex data resources are represented with *DataInstances*. Each *DataInstance* has a *datatype*, representing the *Datatype* that it actualize. There are five subclasses of *DataInstance* class; namely *List*, *Location*, *MediaResource* and *CompositeDataInstance*.

A *List* represents a list of arbitrary items, which can be any literal value or uri. A *List* has *items*, which identifies an RDF List that keeps the items. Figure 3.11 illustrates an example *List* representation. The *List* has a *label* of “My list of things”, a *UserAccount* as its *creator*, and specified to be created at a specific time. It has three items, all having different types.

*MediaResource* represents audiovisual data resources. There are three subclasses of *MediaResource*; namely *Image*, *VideoTrack* and *AudioTrack*. A *MediaResource* has a *datatype* that is either a *MediaType* or a *DerivedDatatype* that has a *MediaType* as an ancestor. *locator* property in Ontology for Media Resources [46] is utilized to identify the URL for the *MediaResources*. Figure 3.12 demonstrates an example *Image* whose datatype is the *DerivedDatatype* represented in Figure 3.3.

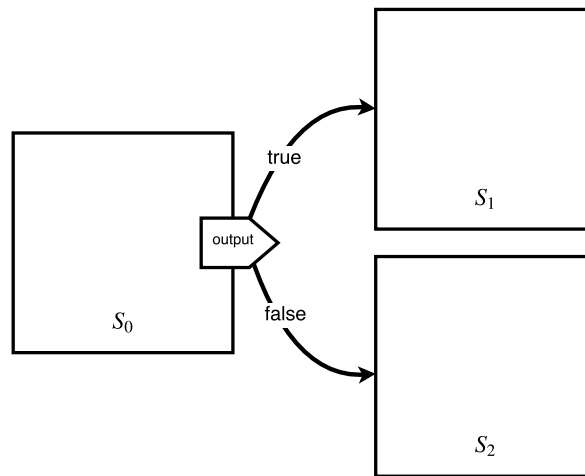


Figure 3.10. An illustration of a *Workflow* with three *Steps* and two *ConditionalPipes*. The *ConditionalPipes* are represented with labeled arrows.

```

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix poc: <http://www.cmpe.boun.edu.tr/soslab/ontologies/poc/core#>.
<http://example.org/myList> a poc:List;
  dcterms:created "2015-12-03T23:48:20Z"^^xsd:dateTime;
  dcterms:creator <http://example.org/myUserAccount>;
  dcterms:title "My list of things"@en;
  poc:items (1 "two" <http://example.org/dummyURI>).

```

Figure 3.11. An example *List* with three items of different types, serialized in Turtle.

```

<http://example.org/mySmallImage> a poc:Image;
  poc:datatype <http://example.org/SmallImageType>;
  ma:locator "http://example.org/anImageURL.jpg"^^xsd:anyURI.

```

Figure 3.12. An example *Image*, represented in Turtle.

Table 3.2: Predefined *Tasks*.

Name	Inputs	Outputs	Description
<i>Create</i>	“object”, “datatype”	“result”	Creates a <i>DataInstance</i> and outputs it with the “result” port. The “object” port identifies the <i>DataInstance</i> whose value is copied into the created one. The “datatype” port identifies the expected <i>Datatype</i> for the “object” port.
<i>Modify</i>	“object”, “value”, “property”, “dataField”	“result”	Modifies the <i>DataInstance</i> identified by the “object” port, inserting the entity identified by the “value” port to the <i>DataInstance</i> . Only one of the “dataField” and “property” ports must be provided. If “dataField” port is present the <i>DataInstance</i> is interpreted to be a <i>CompositeDataInstance</i> , and the value is inserted to it as a <i>fieldValue</i> whose <i>label</i> is identified by the “dataField” port. Otherwise, the property identified by the “property” port is used to insert the value. In either case, the existing values are replaced.
<i>Save</i>	“object”	-	Saves the <i>DataInstance</i> identified by the “object” port, redeeming it referable.
<i>Delete</i>	“object”	-	Deletes the <i>DataInstance</i> identified by the “object” port, redeeming it non-referrable.
<i>Random</i>	“max”	“result”	Generates a random number greater than or equal to zero and less than or equal to the number identified by the “max” port. The generated number has the same literal type as the one identified by the “max” port and is advertised by the “result” port.
<i>Size</i>	“object”	“result”	Outputs the size of the <i>List</i> identified by the “object” port, with the “result” port.
<i>Evaluate</i>	“object”	“result”	Computes the <i>Expression</i> identified with the “object” port, and outputs the result with the “result” port.

Table 3.2: Predefined *Tasks*. (Cont.)

<b>Name</b>	<b>Inputs</b>	<b>Outputs</b>	<b>Description</b>
<i>Insert</i>	“object”, “target”, “index”	“result”	Inserts an item identified by the “object” port to the <i>List</i> identified by the “target” port, and outputs the resulting <i>List</i> with the “result” port. The “index” port expects an integer value and identifies the position in the list for insertion. If no value is provided for the “index” port, the item is appended to the end of the list.
<i>Remove</i>	“object”, “source”, “index”	“result”	Removes an item from the <i>List</i> identified by the “target” port, and outputs the resulting <i>List</i> with the “result” port. The item is either identified directly by the “object” port or with its index by the “index” port.
<i>Get</i>	“source”, “index”	“result”	The “result” port is used to output the item having the index, identified by the “index” port, in the <i>List</i> , identified with the “source” port.
<i>Filter</i>	“object”, “condition”	“result”	Filters the list identified with the “object” port using the <i>Expression</i> identified with the “condition” port, and outputs the resulting list with the “result” port. The <i>Expression</i> is computed for each item of the list, replacing the <code>@item</code> parameter in the <i>Expression</i> . If it computes false or null, the item is excluded.
<i>Identity</i>	-	-	Does not manipulate anything, and is intended to be used for displaying messages to users.

A *CompositeDataInstance* is a *DataInstance*, *datatype* of which is a *CompositeDatatype*. A *CompositeDataInstance* has zero or more *fieldValues*, representing value mappings for the corresponding *DataFields* of its *datatype*. Figure 3.13 depicts a *CompositeDataInstance*, whose *datatype* is the *CompositeDatatype* we defined in Figure 3.4. The illustrated *CompositeDataInstance* has "American haircut" for the "Name" *dataField*, 3 for the "Difficulty" *dataField*, and the *Image* represented in Figure 3.12 for the "Photo" *dataField*. Note that, it does not specify a value for the "Cost" *dataField*, which is allowed to be blank since it is not specified to be required.

```

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix dcterms: <http://purl.org/dc/terms/>.
@prefix poc: <http://www.cmpe.boun.edu.tr/soslab/ontologies/poc/core#>.
<http://example.org/myHaircut> a poc:CompositeDataInstance;
  poc:fieldValue
    [ rdfs:label "Name";
      poc:literalValue "American haircut"
    ],
    [ rdfs:label "Difficulty";
      poc:literalValue 3
    ],
    [ rdfs:label "Photo";
      poc:uriValue <http://example.org/mySmallImage>
    ].

```

Figure 3.13. An example *CompositeDataInstance* with three *fieldValues*, serialized in Turtle.

*WorkflowInstances* are enactments of *Workflows*. Each *WorkflowInstance* has a *datatype*, a *status* and one or more *stepInstances*. The *datatype* of a *WorkflowInstance* is the *Workflow* that is enacted by it. The *status* of a *WorkflowInstance* is a string value. The *stepInstances* of a *Workflow* are the enactments of its individual *Steps*.

A *StepInstance* is an enactment of a *Step*. Each *StepInstance* has a *datatype*, a *status*, zero or one *view*, zero or more *inputs* and zero or more *outputs*. The *view* of a *StepInstance* is a *View*. The *status* of a *StepInstance* is a string value. The *inputs* and *outputs* of a *StepInstance* represent the individual values bound for each *Port* of its *datatype Step*.

### 3.1.5. Representation of Constraints

*Constraints* represent restrictions on when and by whom certain actions can be taken within a POC. Each *Constraint* is associated with a *Workflow*, a *Step* or a *DataInstance*, by *constraint* property. A *Constraint* of a *Workflow* restrains the invocation of the *Workflow* by the users. A *Constraint* of a *Step* restrains its performance by the users. A *Constraint* of a *DataInstance* restrains its access by the users. A *Constraint* is either an *AuthorizationalConstraint* or a *TimeConstraint*. A *TimeConstraint* is a restriction on when a certain action is performable. It either enforces a specific time interval or a lifetime for the availability of its target. *AuthorizationalConstraints* are restrictions on who can perform certain actions. An *AuthorizationalConstraint* can have zero or more *allowedRoles* and zero or more *allowIfs*. An *allowedRole* is a *Role*, possessors of which are allowed by the *Constraint* to perform the restrained action. An *allowIf* is an *Expression* that needs to evaluate true for the allowance of the restrained action. A *Constraint* is deemed as “allowing” if one of its *allowedRoles* or *allowIfs* allows the action. However, all the *Constraints* associated with an action needs to be “allowing” for it to be allowed.

## 3.2. POC LDP Ontology

In this section, we discuss POC LDP Ontology, which is a lightweight extension of POC Core Ontology that provides a vocabulary for the data interchange between POC Applications and POC Clients. POC LDP Ontology is identified by the namespace URI <http://www.cmpe.boun.edu.tr/soslab/ontologies/poc/ldp#>. The namespace prefix `poc-ldp` will be used for this namespace in this document.

Figure 3.14 depicts all the entities defined in POC LDP Ontology. Circle nodes represent OWL 2 classes. The classes that do not belong to poc-ldp namespace are labeled as “external”. Dotted edges represent the assertions in the ontology. Filled edges represent the domain and range constraints for the properties. *POCConcept* and *WorkflowInstance* belong to poc namespace. *Container* represents an LDP Container [38].

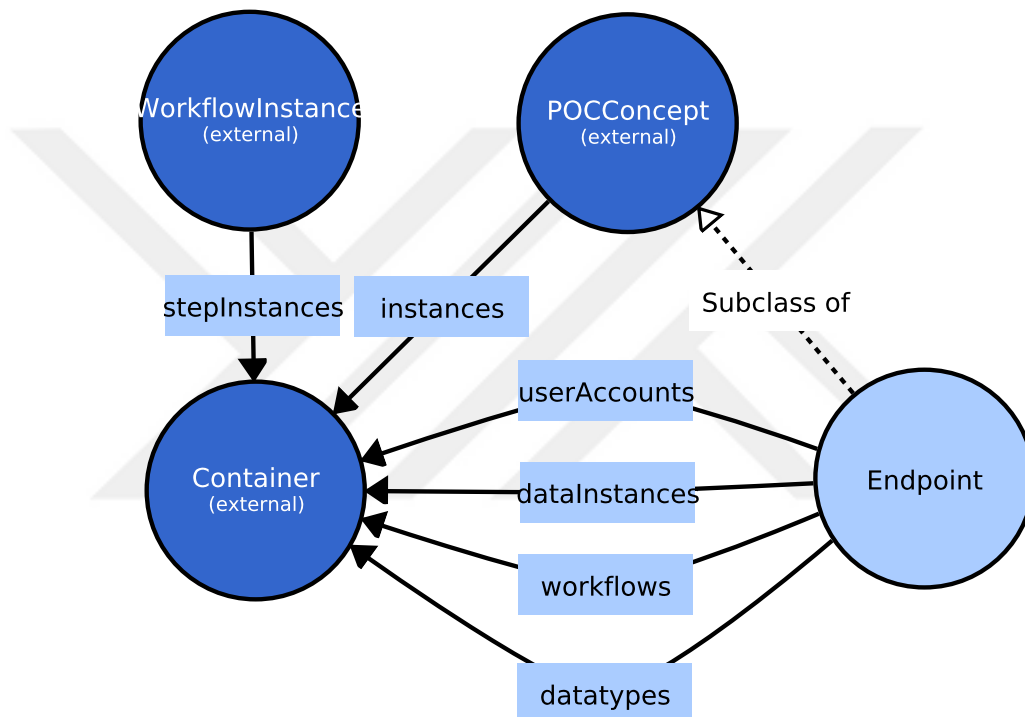


Figure 3.14. A visualization of the classes, properties and assertions in POC LDP Ontology.

An *Endpoint* represents the endpoint of a POC Application, which advertise all information regarding the POC Application. The *dataInstances* property identifies a *Container* that encapsulates the available *DataInstances*. The *userAccounts* property identifies a *Container* that encapsulates the available *UserAccounts*. The *workflows* property identifies a *Container* that encapsulates the available *Workflows*. The *datatypes* property identifies a *Container* that encapsulates the available *Datatypes*.

The *instances* property is used to identify a *Container* that encapsulates the instances of instantiatable entities, such as *Datatypes*, *Workflows* and *Steps*. For instance, a *WorkflowInstance* whose *datatype* is a specific *Workflow*, would be embodied by a *Container* that is the *instances* of that *Workflow*.

The *stepInstances* property identifies a *Container* that encapsulates the *stepInstances* of a *WorkflowInstance*.

### 3.3. Specification of POC Applications

In this section, we discuss the process of specification for applications to be generated.

Initialization of a POC Application requires a specification of its composing elements. The specification process for POC Applications is depicted in Figure 3.15. In the specification process, a description for a POC Application is captured from a Community Builder, and accumulated into a POC Specification.

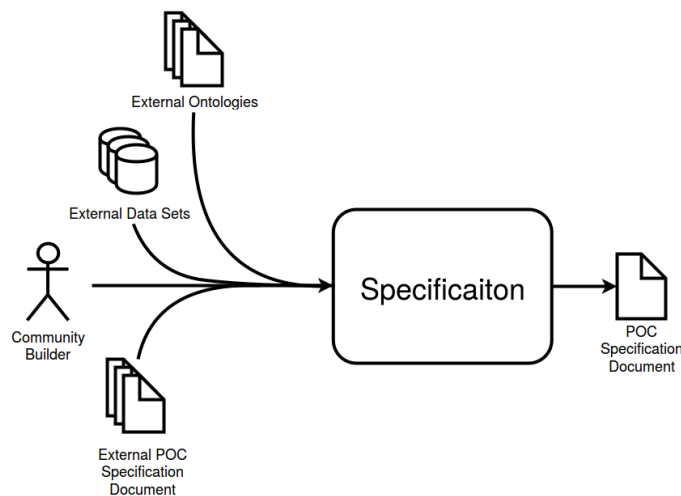


Figure 3.15. An overview of specification process for POC Applications.

POC Model does not provide exact details of how these specifacaitons are captured from human users, but identifies the following requirements for the specification



process.

- (i) People who know the specific needs of individual communities do not necessarily have the technical skills to build applications addressing those needs. The specification process should, therefore, involve an easy-to-use interface, enabling those people to express the features they desire in the applications without manipulating complex source code.
- (ii) The specification process must produce a POC Specification, which reflects the input captured from the Community Builder, and complies with POC Core Ontology.
- (iii) The specification process must allow the import of external ontologies, which may contain entities that are useful for the representation of community data resources.
- (iv) The specification process should provide means for the reuse of structures defined in other POC Specifications, which may be accessible on the Web, or introduced directly by the Community Builder.
- (v) There is a substantial amount of Linked Data [9] accessible on the Web, covering a variety of knowledge domains. Such Linked Data could be useful for POCs, if they have the means to access and utilize it properly. Such data is advertised by open data sets, and available for any type of application. The specification process must allow the Community Builders to introduce Linked Data sets, data from which will be referred by the generated POC Application.

A POC Specification is an explicit specification of the initial configuration of a POC Application. These documents are RDF documents structured in according to POC Core Ontology. A POC Specification typically contain a instances of *UserDefined-Datatypes*, *DataInstances*, *Roles*, *Workflows* and *Constraints* for the POC Application to be constructed. The instances defined within a POC Specification are typically represented with Blank nodes [24] in the process of specification. The rationale behind this will be discussed in Section 3.4.

In the remainder of this section, we introduce a community of bird watchers, referred as  $C_{Birders}$ , and demonstrate how a POC Application serving this community can be defined within a POC Specification.

$C_{Birders}$  is a community that aims to build a catalog of the observations they have made of the birds they encountered in the wild. They would like to document their observations with a photograph they have taken, the location where the photograph was taken, the type of the bird in the photograph, and a descriptive title for their observation.

Here, we demonstrate how a POC Application addressing the needs of  $C_{Birders}$  can be represented by a POC Specification, referred as  $Spec_{Birders}$ . Note that, there can be many acceptable ways to construct an application for a POC. Here, we are putting ourselves in the shoes of a Community Builder, and building a POC Application that we consider to be reasonable for this community.

There are two *Roles* defined in  $Spec_{Birders}$ ; namely, *BirdWatcher* and *BirdExpert*. The representation of these *Roles* in  $Spec_{Birders}$  is demonstrated in Figure 3.16. *BirdWatchers* are ordinary members of  $C_{Birders}$ , who can share their observations with the other members. *BirdExperts* are members who have a trusted expertise to identify the birds shared by the other members.

```
_:Observer a poc:Role;
  rdfs:label "Observer"@en.
_:BirdExpert a poc:Role;
  rdfs:label "Bird Expert"@en.
```

Figure 3.16. The *Roles* in  $Spec_{Birders}$ , serialized in Turtle.

$Spec_{Birders}$  contains a single *UserDefinedDatatype*, which is a *CompositeDatatype* representing the observations of the community members. Figure 3.17 demonstrates

the representation of the datatype, which we refer as *Observation*. *Observation* has "Observation" as its *label*, and the following four *dataFields* that are enumerated by their *labels*.

- "Title": This *DataField* represents the descriptive titles provided for the observations. Its *fieldType* is string.
- "Photograph": This *dataField* represents the photographs the observers have taken of the birds. It has a *fieldType* of *ImageType*.
- "Place": This *DataField* represents the location that the photograph was taken. We want the location data to be retrieved from an open Linked Dataset, so that the geospatial relations of an *Observation* with another *Observation*, or even other external resources, can be identified with well-defined meaning. This *DataField* has a *fieldType* of *Feature*. *Feature* is a class defined in Geonames Ontology [48] that represents any geospatial entity; such as cities, countries, forests, oceans etc. This *DataField* has a *sourceService* identifying a web service where the data for this field will be retrieved from. This service is identified to conform the SPARQL-QUERY protocol, and to be accessible through the endpoint <http://factforge.net/sparql>.
- "Bird Type": This *DataField* represents the type of the observed bird. Unlike the other *DataFields*, it is not specified to be required, indicating that it can be left blank. It has a *fieldType* of *Bird*, which is an entity defined in UMBEL Reference Concept Ontology [37]. This *DataField* has a *sourceService* identifying a web service where the data for this field will be retrieved from. This service is identified to conform the SPARQL-QUERY protocol, and to be accessible through the HTTP endpoint <http://dbpedia.org/sparql>.

The only *DataInstance* defined in *SpecBirders* is *ObservationList*. *ObservationList* is a *List* that contains the *Observations* that are provided by the members. Representation of *ObservationList* in *SpecBirders* is depicted in Figure 3.18.

```

_:Observation a poc:CompositeDatatype;
  rdfs:label "Observation"@en;
  poc:dataField
    [ rdfs:label "Title"@en;
      poc:isRequired true;
      poc:fieldType xsd:string
    ],
    [ rdfs:label "Bird Type"@en;
      poc:fieldType umbel-rc:Bird;
      poc:sourceService
        [ sioc-services:service_protocol "SPARQL-QUERY";
          sioc-services:service_endpoint <http://dbpedia.org/sparql>
        ]
    ],
    [ rdfs:label "Place"@en;
      poc:isRequired true;
      poc:fieldType gn:Feature;
      poc:sourceService [
        sioc-services:service_protocol "SPARQL-QUERY";
        sioc-services:service_endpoint <http://factforge.net/sparql>
      ]
    ],
    [ rdfs:label "Photograph"@en;
      poc:isRequired true;
      poc:fieldType poc:ImageType
    ].

```

Figure 3.17. *Observation* serialized in Turtle.

```

_:ObservationList a poc:List;
  rdfs:label "ObservationList"@en;
  dcterms:title "Observations"@en;
  dcterms:description "This is a list of all the bird observations we
    have gathered so far."@en.

```

Figure 3.18. The representation of *ObservationList*, serialized in Turtle.

*SpecBirders* contains two *Workflows*; namely, *ObservationEntry* and *BirdTypeEditing*. *ObservationEntry* is a *Workflow* that represents the operation of entering *Observations*. Its definition in *SpecBirders* is depicted in Figure 3.20. Figure 3.19 illustrates *ObservationEntry*. It involves the entry of an *Observation* by a community member, and its appending to *ObservationList*. *ObservationEntry* has a *constraint*, which is an *AuthorizationalConstraint* whose *allowedRole* is *BirdWatcher*. This *Constraint* specifies that it is invocable only by *BirdWatchers*.

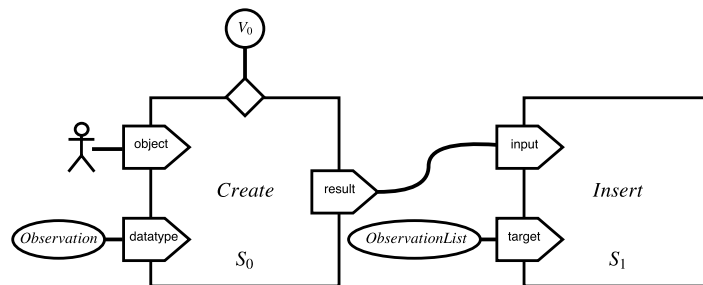


Figure 3.19. A representation of *ObservationEntry*.

*BirdTypeEditing* is a *Workflow* that represents the modification that *BirdExperts* can perform on *Observations*. Its definition in *SpecBirders* is depicted in Figure 3.22. Figure 3.21 illustrates *BirdTypeEditing*. *BirdTypeEditing* enables *BirdExperts* to specify the bird types that are missing in *Observations*, or correct the bird types specified wrongly by *BirdWatchers*. It has a *constraint*, which is an *AuthorizationalConstraint* whose *allowedRole* is *BirdExpert*. This *Constraint* specifies that it is invocable only by *BirdExperts*.

```

_:ObservationEntry a poc:Workflow;
  rdfs:label "Observation Entry"@en;
  poc:constraint [ poc:allowedRole _:BirdWatcher ];
  poc:step _:S0, _:S1;
  poc:pipe [ a poc:HumanPipe; poc:targetStep _:S0; poc:targetPort
    "object"],
    [ a poc:DirectPipe; poc:targetStep _:S0; poc:targetPort
    "datatype"; poc:sourceValue _:Observation],
    [ a poc:PortPipe; poc:sourceStep _:S0 ; poc:sourcePort "result";
    poc:targetStep _:S1 ; poc:targetPort "object"],
    [ a poc:DirectPipe; poc:targetStep _:S1 ; poc:targetPort "target";
    poc:sourceValue _:ObservationList ].

_:S0 a poc:Step;
  poc:task poc:Create;
  poc:viewTemplate "<p>Please submit an
    observation.</p>"^^poc:ViewTemplate;
  poc:inputPort [ rdfs:label "object"],
    [ rdfs:label "datatype"];
  poc:outputPort [ rdfs:label "result"].

_:S1 a poc:Step;
  poc:task poc:Insert;
  poc:inputPort [ rdfs:label "object"],
    [ rdfs:label "target"];
  poc:outputPort [ rdfs:label "result"].

```

Figure 3.20. *ObservationEntry*, serialized in Turtle.

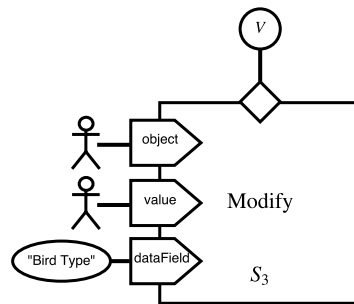


Figure 3.21. A representation of *BirdTypeEditing*.

```

_:BirdTypeEditing a poc:Workflow;
  rdfs:label "Bird Type Editing"@en;
  poc:constraint [ poc:allowedRole _:BirdExpert ].
  poc:step _:S3;
  poc:pipe [ a poc:HumanPipe; poc:targetStep _:S3; poc:targetPort
    "object"],
    [ a poc:HumanPipe; poc:targetStep _:S3; poc:targetPort "value"],
    [ a poc:DirectPipe; poc:targetStep _:S3; poc:targetPort
    "dataField", poc:sourceValue "Bird Type"@en ],
_:S3 a poc:Step;
  poc:task poc:Modify;
  poc:viewTemplate "<p>Please select a bird type for this
    observation.</p>"^^poc:ViewTemplate;
  poc:inputPort [ rdfs:label "object"], [ rdfs:label "value"], [
    rdfs:label "dataField"].

```

Figure 3.22. A representation of *BirdTypeEditing* in *SpecBirders*, serialized in Turtle.

### 3.4. Execution of POC Applications

In this section, we discuss the execution process in POC Model for applications that actualize POC Specifications, which are called POC Applications. First, we give an overview of the execution process for POC Applications. Then, we provide a list of requirements we identified for POC Applications. Lastly, we discuss the architecture and execution behaviour of POC Applications.

An overview of the execution process in POC Model is depicted in Figure 3.23. The execution process involves the initialization of a POC Application that conforms a POC Specification, and its communication with the members of a POC who may use various client applications. Initialization of a POC Application and its runtime behaviour is depended on the semantics defined within POC Core Ontology. The communication of a POC Application with its users is structured by POC LDP Ontology, which is an extension of POC Core Ontology.

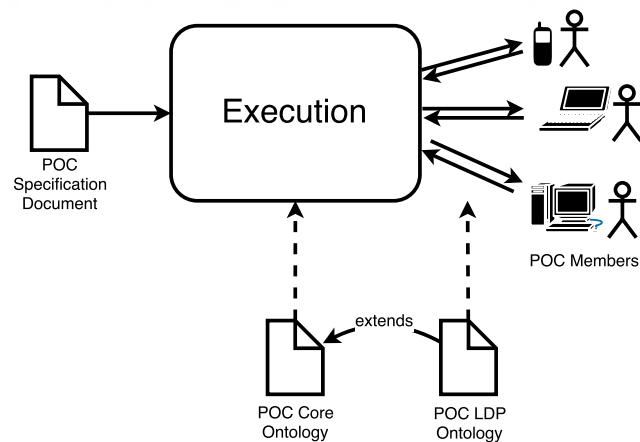


Figure 3.23. An overview of the execution process in POC Model.

We identify the following requirements for a POC Application.

- (i) A user shall be able to join the associated POC through the system.
- (ii) A user shall be able to view the community data resources.



- The system shall only allow access to a data resource if the requesting user satisfies the authorizational constraints associated with the data resource.
  - The system shall only allow access to a data resource if the time constraints associated with the data resource are satisfied.
- (iii) A user shall be able to view and invoke workflows.
- The system shall only allow access and invocation of a workflow if the requesting user satisfies the authorizational constraints associated with the workflow.
  - The system shall only allow access and invocation of a workflow if the time constraints associated with the workflow are satisfied.
- (iv) The user shall be able to view the status of the workflow instances that s/he invoked.
- (v) A user shall be able to view and perform tasks.
- The system shall only allow access and performance of a task if the requesting user satisfies the authorizational constraints associated with the task.
  - The system shall only allow access and performance of a task if the time constraints associated with the task are satisfied.
  - The system shall allow access and performance of a task only if all of its control and data dependencies are satisfied.
  - The system shall present the data utilized within the task.
  - The system shall provide instructions on how to perform the task.
  - The system shall provide means for the entry of the data required for the execution of the task.
- (vi) The user shall be able to view the information of a task that s/he previously performed.
- (vii) The system shall handle the execution of the tasks.
- The system shall execute the tasks that do not depend on human input automatically when their control and data dependencies are satisfied.

In the remainder of this section, we discuss the architectural design and execution details of POC Applications, and discuss how they address the listed requirements.

POC Applications follow the client-server architectural design pattern, which entails a separation of the server-side concerns, such as persistence or business logic, and the client-side concerns, such as the user interface. The server-side component of a POC Application is called a POC Server. A client-side applicaiton that interacts with a POC Server is called a POC Client.

POC Applications have a stateless architecture. This indicates that a POC Server must be able to interpret any request properly without the information of previous requests. POC Model does not assert specific authentication and authorization mechanisms, however, any implemented mechanism must not interfere with the statelessness. This necessitates POC Clients to keep track of the session state.

The communication between POC Servers and POC Clients conform Linked Data Platform 1.0 (LDP) Specification [38], where POC Servers are LDP Servers and POC Clients are LDP Clients. A POC Server and the resources it advertise are identified by dereferenceable HTTP URIs, and represented as RDF data, structured with POC Core Ontology and POC LDP Ontology.

The remaining discussion of POC Applications in this section include demonstrative interchanges between POC Servers and POC Clients, for an example POC Application that we will refer as *AppBirders*. *AppBirders* conforms *SpecBirders*, which is the POC Specification discussed in the previous section. The POC Server of *AppBirders* is assumed to be identified by the URI `http://example.org/birders`. For the sake of brevity, the HTTP headers interchanged through the following HTTP transactions are mostly omitted, unless they play an important role for the model. Also, for readability, all the provided examples interchange RDF data only in Turtle. Although, POC Servers must also support JSON-LD, as required by LDP.

### 3.4.1. Initialization of a POC Server

InitIALIZATION of a POC Server typically involves the setup of a database environment to satisfy the persistence needs of the POC Application, a web server to

communicate with the clients, and any other ancillary construct, such as a load balancer or a cache store. POC Model does not assert constraints on the inner architecture of the POC Servers, nonetheless, it specifies their communication with their clients.

As a POC Server is initialized, it generates and assigns a URI to each community-specific element that is identified by a blank node in the POC Specification, such as workflows, steps, datatypes, roles, data instances. Such URIs are needed for the unambiguous identification of the community elements by POC Clients. POC Model does not impose a specific rule on the structure of the generated URIs, yet, it is advisable to implement a URI generation mechanism that produce human-readable URIs. The subsequent discussion on the execution model includes examples on such URIs.

### 3.4.2. Retrieval of a POC Server's Representation

When dereferenced through its URI, a POC Server guides the client on where to find the available resources within the application. Such resources include the community workflows, datatypes, data instances and user accounts if they exist. A POC Server may advertise additional properties along with the mentioned elements. Figure 3.24 depicts an HTTP request for retrieving a representation of the POC Server of *AppBirders*.

```
GET /birders HTTP/1.1
Host: example.org
Accept: text/turtle
```

Figure 3.24. An HTTP request message for dereferencing the POC Server of *AppBirders*.

The corresponding HTTP response is demonstrated by Figure 3.25. The response message indicates that the requested resource is a POC Server, and identifies the URIs

of four LDP Basic Containers that instore various types of community resources. A container of the available community workflows is identified via *poc-ldp:workflows*. A container of the available data resources is identified via *poc-ldp:dataInstances*. A container of the community datatypes is identified via *poc-ldp:datatypes*. A container of the user accounts of the community members is identified via *poc-ldp:userAccounts*. A POC Client may navigate through the provided URIs automatically, or expect input from its user before submitting any additional requests.

```

HTTP/1.1 200 OK
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD

<http://example.org/birders> a poc-ldp:POC_Server;
    poc-ldp:dataInstances <http://example.org/birders/data_instances/>;
    poc-ldp:datatypes <http://example.org/birders/datatypes/>;
    poc-ldp:workflows <http://example.org/birders/workflows/> .

```

Figure 3.25. An HTTP response message advertising the POC Server of *AppBirders*.

### 3.4.3. Retrieval of the Community Datatypes

A POC Server advertise the custom datatypes that structure the community resources as elements of an LDP Basic Container. The URI of this container is identified with *poc-ldp:datatypes* property in the representation of a POC Server, as demonstrated in Section 3.4.2. An HTTP request message for the retrieval of the community datatypes for *AppBirders* is demonstrated by Figure 3.26. The corresponding HTTP response from the POC Server is depicted by Figure 3.27. The response message indicates that the requested resource is an LDP Basic Container, with a single element that is identified by the relative URI *<observation>*. There may be other datatypes structuring the data resources within *AppBirders*, however, this is the only community-specific datatype.

```

GET /birders/datatypes/ HTTP/1.1
Host: example.org
Accept: text/turtle

```

Figure 3.26. An HTTP request message for the retrieval of the community-specific datatypes of *AppBirders*.

```

HTTP/1.1 200 OK
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD

<> a ldp:BasicContainer;
  ldp:contains <observation>.

<observation> a poc:CompositeDatatype;
  rdfs:label "Observation"@en.

```

Figure 3.27. An HTTP response message advertising the community-specific datatypes of *AppBirders*.

An HTTP request to retrieve the advertised datatype is depicted by Figure 3.28.

The corresponding HTTP response from the POC Server is demonstrated in Figure 3.29. The returned data is nearly identical to the specification of *Observation* depicted by Figure 3.17 in Section 3.3, the only difference being that it is now identified by a URI, rather than a blank node.

In order to interpret the data resources advertised by a POC Server, a POC Client may also need the information of the datatypes defined within POC Core Ontology or

```

GET /birders/datatypes/observation HTTP/1.1
Host: example.org
Accept: text/turtle

```

Figure 3.28. An HTTP request message for the retrieval of a community-specific *Datatype*.

other external sources. Such datatypes should be dereferencable through their URIs, so that they can be retrieved by POC Clients automatically.

#### 3.4.4. Retrieval and Invocation of the Community Workflows

A POC Server advertises the community workflows available for the user as the elements of an LDP Basic Container, addressing the aforementioned Requirement iii of POC Applications. The URI of this container is identified with *poc-ldp:workflows* property in the representation of a POC Server, as demonstrated in Section 3.4.2. An HTTP request message for the retrieval of the available workflows for a user of *AppBirders* is given in Figure 3.30.

The corresponding HTTP response is demonstrated in Figure 3.31. The user who initiates the depicted transaction is assumed to possess the role *BirdWatcher* and not *BirdExpert*, and therefore has access only to *ObservationEntry* workflow. The relative URIs in the response are based on the URI of the requested resource, i.e. `<>` represents `<http://example.org/birders/workflows/>` and `<observation_entry>` represents `<http://example.org/birders/workflows/observation_entry>`. The response message indicates that the requested resource is an LDP Basic Container, with a single element identified by the relative URI `<observation_entry>`. A representation of this element as well is included in the response message. This element is identified to be a workflow, and is *labeled* as "Observation Entry".

```

<> a poc:CompositeDatatype;
  rdfs:label "Observation"@en;
  poc:dataField
    [ rdfs:label "Title"@en;
      poc:isRequired true;
      poc:fieldType xsd:string
    ],
    [ rdfs:label "Bird Type"@en;
      poc:fieldType umbel-rc:Bird;
      poc:sourceService
        [ sioc-services:service_protocol "SPARQL-QUERY";
          sioc-services:service_endpoint <http://dbpedia.org/sparql>
        ]
    ],
    [ rdfs:label "Place"@en;
      poc:isRequired true;
      poc:fieldType gn:Feature;
      poc:sourceService [
        sioc-services:service_protocol "SPARQL-QUERY";
        sioc-services:service_endpoint <http://factforge.net/sparql>
      ]
    ],
    [ rdfs:label "Photograph"@en;
      poc:isRequired true;
      poc:fieldType poc:ImageType
    ].

```

Figure 3.29. An HTTP response message advertising a representation of a community-specific *Datatype*.

```

GET /birders/workflows/ HTTP/1.1
Host: example.org
Accept: text/turtle

```

Figure 3.30. An HTTP request message for the retrieval of the available workflows in

*AppBirders*.

```

HTTP/1.1 200 OK
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD

<> a ldp:BasicContainer;
  ldp:contains <observation_entry>.

<observation_entry> a poc:Workflow;
  rdfs:label "Observation Entry"@en.

```

Figure 3.31. An HTTP response message for the retrieval of the available workflows

in *AppBirders*.

POC Servers are not required to return the representations of individual elements along with the containers, however, they may do so in order to relieve the clients of submitting additional requests for their retrieval. A POC Client may utilize HTTP Prefer header [49] to cue POC Servers on the desired verbosity for the representation of LDP Containers, as described by LDP [38]. If the representation of the workflow were not included in the response message above, the requesting POC Client would need to retrieve its representation via an additional request, as demonstrated in Figure 3.32.



```

GET /birders/workflows/observation_entry HTTP/1.1
Host: example.org
Accept: text/turtle

```

Figure 3.32. An HTTP request message for the retrieval of *ObservationEntry*.

The corresponding HTTP response message is demonstrated in Figure 3.33. The POC Server answers with an HTTP response message that includes POST as an allowed HTTP method, implying that the user is allowed to perform a POST request against the requested URI [40].

```

HTTP/1.1 200 OK
Content-Type: text/turtle
Allow: POST,GET,OPTIONS,HEAD

<> a poc:Workflow;
  rdfs:label "Observation Entry"@en.

```

Figure 3.33. An HTTP response message advertising *ObservationEntry*.

A POST request against the URI of a workflow creates an instance of that workflow, or in other words, invokes the workflow. Figure 3.34 demonstrates an HTTP request message for such an invocation of *ObservationEntry*.

The corresponding HTTP response message is demonstrated by Figure 3.35. The POC Server responds with the HTTP status code 201, indicating that one or more new resources were created as a result of the request [50]. The response message includes a Location header, identifying the URI of the resource that has been created as a result of the transaction. A representation of the created resource is included in the message body. The advertised resource is a *WorkflowInstance* whose *datatype*

```

POST /birders/workflows/observation_entry HTTP/1.1
Host: example.org
Accept: text/turtle

```

Figure 3.34. An HTTP request message invoking *ObservationEntry*.

is *ObservationEntry*. The status of the *WorkflowInstance* is "ongoing", indicating that it was invoked, but has not been completed yet. It advertises a single associated *StepInstance* that is available for the requesting user, identified by the URI `http://example.org/birders/step_instances/si0`.

```

HTTP/1.1 201 Created
Location: http://example.org/birders/workflows/observation_entry/oe0
Content-Type: text/turtle

<> a poc:WorkflowInstance;
  poc:datatype <http://example.org/workflows/observation_entry>;
  poc:status "ongoing";
  dcterms:created "2015-12-03T23:48:20Z"^^xsd:dateTime;
  dcterms:creator <http://example.org/birders/user_accounts/ua0>;
  poc-ldp:step_instances <#step_instances>.

<#step_instances> a ldp:BasicContainer;
  ldp:contains <http://example.org/birders/step_instances/si0>.

```

Figure 3.35. HTTP response message for the invocation of *ObservationEntry*.

### 3.4.5. Retrieval and Performance of Step Instances

A POC Server advertise the *StepInstances* available for the user as elements of an LDP Basic Container. The URI of this container is identified with *poc-ldp:stepInstances* property in the representation of a POC Server, as demonstrated in Section 3.4.2. The presented *StepInstances* may belong to a *WorkflowInstance* invoked by the user, or others. An HTTP request for retrieving the *StepInstances* available for a user of *AppBirders* is demonstrated by Figure 3.36.

```
GET /birders/step_instances/ HTTP/1.1
Host: example.org
Accept: text/turtle
```

Figure 3.36. An example HTTP request message for retrieving the *StepInstances* available for a user of *AppBirders*.

The corresponding HTTP response message is demonstrated by Figure 3.35. The response message states that the requested resource is an LDP Basic Container, with three elements identified by the relative URIs *<si0>*, *<si1>* and *<si2>*.

```
HTTP/1.1 200 OK
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD

<> a ldp:BasicContainer;
  ldp:contains <si0>, <si1>, <si2>.
```

Figure 3.37. An example HTTP response message advertising the *StepInstances* available for a user of *AppBirders*.

An HTTP request message for retrieving one of the identified *StepInstances* is depicted in Figure 3.38. We will refer to this *StepInstance* as  $SI_0$ .

```
GET /birders/step_instances/si0 HTTP/1.1
Host: example.org
Accept: text/turtle
```

Figure 3.38. An HTTP request retrieving a representation of  $SI_0$ .

The corresponding HTTP response message is demonstrated by Figure 3.39. The response message includes PATCH as an allowed HTTP method, indicating that the requester is allowed to perform a PATCH request against the requested URI [40]. A representation  $SI_0$  is provided by the response message. The *status* of  $SI_0$  is "ready", indicating that it is available for the user to perform.  $SI_0$  has a *view* to be displayed to the user by the client. The *Step* that is the *datatype* of  $SI_0$  is identified, which we will refer to as  $S_0$ . A representation of  $S_0$  is also included in the response.  $S_0$  has *Create* as its *task*.  $S_0$  has two *inputPorts* labeled as "datatype" and "object", informing the client on the inputs required for performing  $SI_0$ .  $SI_0$  has an *input* labeled "datatype" with *Observation* as its value. This indicates that the value for the corresponding *InputPort* of  $S_0$ , which is labeled as "datatype", is given to be *Observation*, and that the value for the other *InputPort*, which is labeled as "object", should be provided by the client for the execution of  $SI_0$ . *Observation* is a *CompositeDatatype* defined within *SpecBirders*. The specification of *Observation* is provided by Figure 3.17 in Section 3.3, and its retrieval by a POC Client is demonstrated by Figure 3.28 and 3.29 in Section 3.4.3.

A POC Client that has retrieved a representation of a *StepInstance* with "ready" status must prompt the user with a Web form to capture the data required for the execution of the *StepInstance*. Such a Web form needs to be structured in according to the types of expected input types, and may need to fetch data from external data sources and display them to the user as candidate input values. Such information

```

HTTP/1.1 200 OK
Content-Type: text/turtle
Allow: PATCH,GET,OPTIONS,HEAD

<> a poc:StepInstance;
  poc:datatype
    <http://example.org/birders/workflows/observation_entry/steps/s0>;
  dcterms:created "2015-12-03T23:48:20Z"^^xsd:dateTime;
  dcterms:creator <http://example.org//user_accounts/ua0>;
  poc:status "ready";
  poc:view "<p>Please submit an observation.</p>";
  poc:input [ rdfs:label "datatype";
    poc:value <http://example.org/birders/datatypes/observation> ].

<http://example.org/birders/workflows/observation_entry/steps/s0> a
  poc:Step;
  poc:task poc:Create;
  poc:inputPort [ rdfs:label "datatype"],
    [ rdfs:label "object"].

```

Figure 3.39. An HTTP response message advertising  $SI_0$ .

is provided by the *Step* that is the *datatype* of the *StepInstance* and the *Datatypes* associated with the *inputs* of the *StepInstance*. These resources can be advertised within the representations of *StepInstances* by a POC Server, just as the representation of  $SI_0$  includes a specification of  $S_0$ . Otherwise, they should be retrieved via additional requests by POC Clients, and cached for future references. A POC Client that has received the representation of  $SI_0$  above, for instance, should retrieve *Observation*, unless it is already fetched and cached.

A Web form capturing the human inputs for a *StepInstance* must provide means for the submission of numerous types of data. Data structured with most of such types, such as string, integer, image, can be captured through standard HTML form elements. Handling *CompositeDatatypes*, such as *Observation*, are not as straightforward. *CompositeDatatypes* typically consist of multiple *DataFields*, each of which may have different, and possibly external, types. A POC Client should discover and fetch any of such types, and construct the form accordingly. For a *DataField* associated with a *sourceService*, the POC Client must ensure that the provided values for the *DataField* are selected from the data advertised by that service. A POC Client that initiates the execution of  $SI_0$  in *AppBirders*, for example, would need to implement such a mechanism for the *DataFields* labeled "Bird Type" and "Place" of *Observation*. POC Model does not impose a specific means for the retrieval and presentation of candidate values for such *DataFields*, nevertheless, a text auto-completion mechanism that suggest candidate values based on text entry could be a reasonable solution for many cases. For, say, the "Bird Type" *DataField* of *Observation*, a POC Client could provide a text field for the entry of a bird species and use the entered text to fetch and suggest candidate values to the user. To fetch the candidate values, the POC Client needs to send a request to the specified endpoint of the *sourceService* in the specified protocol. The endpoint for the *sourceService* of the "Bird Type" *DataField* is <http://dbpedia.org/sparql> and its protocol is SPARQL-QUERY [29]. If the user entered "sto" to the form field for the "Bird Type" *DataField*, for instance, the POC Client could submit the SPARQL query demonstrated by Figure 3.40 to <http://dbpedia.org/sparql>.

The service endpoint responds with a list of results as depicted in Table 3.3. The POC Client that has received such data should, then, display the returned labels to the user as candidates for the *DataField*. If the user selects one of the candidate labels, the corresponding resource URI should be selected to be submitted for the "Bird Type" *DataField* of *Observation*.

Performance of a *StepInstance* by a user requires the submission of the required input data by a POC Client. Such submissions are enacted through PATCH requests against the URI of the *StepInstance*, attaching input values to the *StepInstances*. The

```

SELECT DISTINCT ?bird ?label WHERE
{
  ?bird a umbel-rc:Bird.
  ?bird rdfs:label ?label.
  FILTER regex(?label, "^sto.*", "i")
} LIMIT 5

```

Figure 3.40. An example SPARQL query to fetch five bird species whose labels start with "sto".

patching of the *StepInstances* conforms Linked Data Patch Format (LD Patch) [39], which is the patching mechanism for Linked Data resources favoured by LDP [38]. An HTTP request for such a submission for  $SI_0$  is demonstrated by Figure 3.41. The request message utilize the Add operation of Linked Data Patch Format to insert an input to  $SI_0$ . The inserted input is labeled as "object", and has a *CompositeDataInstance* as its value. The *CompositeDataInstance* has *Observation* as its *datatype* and four *fieldValues*, each corresponding to a different *DataField* of *Observation*.

Table 3.3. The results for the SPARQL query fetching bird species and their labels.

bird	label
<a href="http://dbpedia.org/resource/Stork">http://dbpedia.org/resource/Stork</a>	"Stork"@en
<a href="http://dbpedia.org/resource/Stork-billed_Kingfisher">http://dbpedia.org/resource/Stork-billed_Kingfisher</a>	"Stork-billed Kingfisher"@en
<a href="http://dbpedia.org/resource/Stout-billed_Cuckoo-shrike">http://dbpedia.org/resource/Stout-billed_Cuckoo-shrike</a>	"Stout-billed Cuckoo-shrike"@en
<a href="http://dbpedia.org/resource/Stolid_Flycatcher">http://dbpedia.org/resource/Stolid_Flycatcher</a>	"Stolid Flycatcher"@en
<a href="http://dbpedia.org/resource/Storm's_Stork">http://dbpedia.org/resource/Storm's_Stork</a>	"Storm's Stork"@en

```

PATCH /birders/step_instances/si0 HTTP/1.1
Host: example.org
Content-Type: text/ldpatch
Accept: text/turtle

Add { <> poc:input [
  rdfs:label "object";
  poc:value [ a poc:CompositeDataInstance;
    poc:datatype <http://example.org/datatypes/observation>;
    fieldValue
      [ rdfs:label "Title"; poc:literalValue "Stork drinking water." ],
      [ rdfs:label "Bird Type"; poc:uriValue
        <http://dbpedia.org/resource/Stork> ],
      [ rdfs:label "Place"; poc:uriValue
        <http://dbpedia.org/resource/London> ],
      [ rdfs:label "Photograph";
        poc:uriValue [ a poc:Image;
          ma:locator "http://oi67.tinypic.com/254vyh3.jpg"^^xsd:anyURI ]]
    ]
  ].}.

```

Figure 3.41. An HTTP PATCH request submitting the data for the execution of  $SI_0$ .

The corresponding HTTP response message is demonstrated by Figure 3.42. The response advertise a representation of the  $SI_0$ , which now has a "completed" *status*.

### 3.4.6. Retrieval of the Data Resources

A POC Server advertise the data resources available for the user as the elements of an LDP Basic Container. The URI of this container is identified with *poc-*



```

HTTP/1.1 200 OK
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD

<> a poc:StepInstance;
  poc:datatype
    <http://example.org/birders/workflows/observation_entry/steps/s0>;
  dcterms:created "2015-12-03T23:48:20Z"^^xsd:dateTime;
  dcterms:creator <http://example.org/user_accounts/ua0>;
  poc:performedAt "2015-12-03T23:51:04Z"^^xsd:dateTime;
  poc:performer <http://example.org/user_accounts/ua0>;
  poc:status "completed".

```

Figure 3.42. An HTTP response message, advertising the successfully performed  $SI_0$ .

*ldp:dataInstances* property in the representation of POC Servers, as demonstrated in Section 3.4.2. Figure 3.43 depicts an HTTP request message retrieving the data resources available for a user of *AppBirders*.

```

GET /birders/data_instances HTTP/1.1
Host: example.org
Accept: text/turtle

```

Figure 3.43. An HTTP request message, retrieving the data resources available for a user of *AppBirders*.

A successful HTTP response for the request above is depicted by Figure 3.44. The response message contains a representation of an LDP Basic Container, with a single element identified by the URI `http://example.org/birders/data_instances/observations`.

```

HTTP/1.1 200 OK
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD

<> a ldp:BasicContainer;
  ldp:contains </observations>.

```

Figure 3.44. An HTTP response message, advertising the data resources available for a user of *AppBirders*.

A POC Server may advertise the representation of individual data resources along with the container that contains them. Otherwise, a POC Client needs to submit an additional request for their retrieval. Figure 3.45 demonstrates such a request for the retrieval of the *DataInstance* advertised by the server.

```

GET /birders/data_instances/observations HTTP/1.1
Host: example.org
Accept: text/turtle

```

Figure 3.45. An HTTP request message, retrieving an available *DataInstance* of *AppBirders*.

The POC Server for *AppBirders* responds with a message as demonstrated by Figure 3.46. The response message contains a representation of a *List*, with a single item identified by the relative URI <o0>. A representation of the item is also included in the message, which is a *CompositeDataInstance*. The *datatype* of this *DataInstance* is specified to be *Observation*, and it has four *fieldValues*, each corresponding to a different *DataField* of *Observation*.

```

HTTP/1.1 200 OK
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD
<> a poc:List;
  poc:items <o0>.
<o0> a poc:CompositeDataInstance;
  poc:datatype <http://example.org/datatypes/observation>;
  fieldValue
    [ rdfs:label "Title"; poc:literalValue "Stork drinking water." ],
    [ rdfs:label "Bird Type"; poc:uriValue
      <http://dbpedia.org/resource/Stork> ],
    [ rdfs:label "Place"; poc:uriValue
      <http://dbpedia.org/resource/London> ],
    [ rdfs:label "Photograph";
      poc:uriValue [
        a poc:Image;
        ma:locator "http://oi67.tinypic.com/254vyh3.jpg"^^xsd:anyURI
      ]
    ].

```

Figure 3.46. An HTTP response message, advertising a *DataInstance* of *AppBirders*.

### 3.4.7. Execution Flow of Step Instances

In this section, we discuss how interdependent *StepInstances* are to be handled by POC Servers.

The community-centric processes to be performed within a POC are represented by its *Workflows*. *Workflows* are composed of interdependent *Steps* which are either automated or human-based. A *Step*,  $S_0$ , is said to be dependent on another *Step*,  $S_1$ ,

if there is a *Pipe* whose *sourceStep* is  $S_1$  and *targetStep* is  $S_0$ .

Invocation of a *Workflow* yields the creation of a *WorkflowInstance* whose *datatype* is that *Workflow*. Creation of a *WorkflowInstance* involves the creation of a *StepInstance* for each *Step* of the invoked *Workflow*. A *StepInstance*,  $SI_0$ , is said to be dependent on another *StepInstance*,  $SI_1$ , if the *Step* that is the *datatype* of  $SI_0$  is dependent on the *Step* that is the *datatype* of  $SI_1$ . This dependence implies that  $SI_0$  cannot be executed unless the execution of  $SI_1$  is successfully completed.

A *StepInstance* is said to be ready, when all the *StepInstances* it is dependent on have been executed, and thus, have a "completed" status. When a *StepInstance* becomes ready, it is either executed immediately if it is automated, or assigned with a "ready" status if it expects human input. If all the *StepInstances* of a *WorkflowInstance* have "completed" as their status the *WorkflowInstance* must be assigned a "completed" status as well.

### 3.4.8. Expressions and View Templates

*Expressions* resemble programming language expressions [51,52], yet have a much limited expressivity. An *Expression* can contain values, constants, arithmetic operators, comparison operators, and a set of basic methods. The constants in an *Expression* are identified with a string of alphanumeric characters prepended with an @ character, e.g. "@myConstant < @c0 + 4 && @c0 > 2". *Expressions* support the following methods that provide access to information regarding *Instances*.

- Field getter: This method enables the retrieval of *fieldValues* of *CompositeDataInstances* in an *Expression*, e.g. "@myHaircut['price']" returns the *fieldValue* labeled "price" of a *CompositeDataInstance*, given that it is available through "@myHaircut" constant.
- Size: This method returns the size of a *List*, e.g. "@list.size", given that "@list" constant is associated with a *List*.

- Creator: This method returns the *creator* of an *Instance*, e.g. "`@myInstance.creator`", given that the constant "`@myInstance`" is associated with an *Instance*.
- Created: This method returns the creation time of an *Instance*, e.g. "`@myInstance.created`", given that the constant "`@myInstance`" is associated with an *Instance*.

*ViewTemplates* are HTML [53] formatted strings that can contain embedded *Expressions*. The content between the "`{#`" and "`#}`" tags in a *ViewTemplate* are treated as *Expressions*. For instance, "`<img src={#@myImage.locator#}>`" is a *ViewTemplate* that can be used to prompt an image to the user.

*Views* are similar with *ViewTemplates*, as they are HTML [53] formatted strings as well. They, however do not allow embedding of *Expressions*. When a *Step* is invoked, the embedded references in its *ViewTemplate* are replaced with their values, to generate a *View*. The resulting *View* is associated with the *StepInstance* that enacts the *Step*.

## 4. EVALUATION

In this chapter, we discuss our efforts to evaluate this thesis work.

First, we examine the ability of POC Core Ontology to express a set of behavioral features commonly seen in community applications. We, then, evaluate the convenience of POC Model for real world applications. This investigation involves the introduction of an existing POC and the examination of the capability of POC Model to satisfy the specific needs of this community. Lastly, we introduce the proof-of-concept prototype we developed to investigate and demonstrate the integrity of the model.

### 4.1. Utilization of POC Model for a Real World Community

In this section, we introduce a real world example for POCs, and discuss the competence of POC Model to satisfy the needs of this POC.

Oy ve Ötesi is a volunteer organization that aims to increase the transparency of the public elections in Turkey. Prior to the elections, Oy ve Ötesi provides an online platform for the volunteers to track the number of required observers in ballot stations. The volunteers can sign up as observers where needed, and inspect the activities of the officers in each ballot station.

Oy ve Ötesi recently introduced an additional mechanism to improve the transparency of the elections, which is called Türkiye Tutanak Takip<sup>1</sup> (T3). T3 is a human computation application that aims to validate the consistency of the officially announced vote distributions with the vote counts reported by the volunteer observers. Observers of each ballot station submit photographs of the ballot reports they observed to the T3 platform. Each photograph is, then, displayed to random volunteer verifiers who sign up to the T3 platform. The verifiers fill web forms with the information of vote distributions depicted in the photographs. If three verifiers enter the exactly same vote

---

<sup>1</sup>In English: Turkey Report Tracking.

distribution for a ballot box, the ballot report is considered as verified. The difference between the official results and the T3 verified vote counts are then published publicly. T3 was utilized for the first time in Turkish general elections of 2015, and reported [54] to have verified more than 180 thousand ballot reports.

We observed that T3 platform is subject to the following key requirements.

- (i) Volunteers who are assigned as observers should be able to submit photographs.
- (ii) People who are not assigned as observers should not be able to submit photographs.
- (iii) A volunteer who wants to verify ballot reports should be prompted with a random photograph that was previously submitted and a web form that captures the information depicted in the photograph.
- (iv) If there is no photograph to be verified on the platform, the verifying volunteers should be informed so.
- (v) Information of the verified ballot reports should be publicly available, therefore, unauthenticated users of the platform should have access to them.

In the remainder of this section, first we provide a POC Specification for a T3 platform, which verifies the ballot reports of an election between three hypothetical political parties; namely,  $P_A$ ,  $P_B$  and  $P_C$ . Then, we demonstrate the runtime behaviour of a POC Application conforming the provided POC Specification. Lastly, we provide an assessment of how well the constructed application addresses the requirements of the T3 community.

#### 4.1.1. A POC Specification for T3

Here, we construct a POC Specification for T3, which will be referred as  $Spec_{T3}$ .

There are two *Role* roles defined within  $Spec_{T3}$ ; namely, *Observer* and *Verifier*. The representation of these roles in  $Spec_{T3}$  is demonstrated in Figure 4.1. *Observer* represents the observers at the ballot stations, who upload the photographs of ballot

reports. *Verifier* represents the volunteers, who fill web forms with the information of vote distributions depicted in the photographs.

```
_:Observer a poc:Role;
  rdfs:label "Observer"@en.
_:Verifier a poc:Role;
  rdfs:label "Verifier"@en.
```

Figure 4.1. The *Roles* defined in *Spec<sub>T3</sub>*, serialized in Turtle.

*Spec<sub>T3</sub>* contains two *UserDefinedDatatype* definitions; namely, *Photo* and *VerifierReport*.

*Photo* represents the photographs of the ballot reports that are uploaded by *Observers*. Specification of *Photo* within *Spec<sub>T3</sub>* is depicted by Figure 4.2. It is a *DerivedDatatype* with a *baseDatatype* of *ImageType*. As these photographs contain handwritten information to be interpreted by humans, they need to be large enough to be readable. We, thereby, arbitrarily define the minimum frame height as 600 pixels and the minimum frame width as 400 pixels for *Photo*.

```
_:Photo a poc:DerivedDatatype;
  rdfs:label "Photo"@en;
  poc:baseDatatype poc:ImageType;
  poc:minFrameWidth 400;
  poc:minFrameHeight 600.
```

Figure 4.2. Definition of *Photo* in *Spec<sub>T3</sub>*, serialized in Turtle.

*VerifierReport* represents the information provided by the *Verifiers* for each *Photo* they are prompted. Figure 4.3 demonstrates the definition of *VerifierReport* within *Spec<sub>T3</sub>*. *VerifierReport* is a *CompositeDatatype* with four *dataFields*. The first



*DataField* is labeled "ID", and represents the unique ID of a ballot report. The other three *DataFields* are labeled "Votes for A", "Votes for B" and "Votes for C", and represent the vote counts for the political parties  $P_A$ ,  $P_B$  and  $P_C$  respectively. Each *DataField* has a *datatype* of integer and identified to be required, indicating that they cannot be left blank.

```
_:VerifierReport a poc:CompositeDatatype;
  rdfs:label "Verifier Report"@en;
  poc:dataField [ rdfs:label "ID";
    poc:isRequired true;
    poc:fieldType xsd:integer ],
  [ rdfs:label "Votes for A";
    poc:isRequired true;
    poc:fieldType xsd:integer ],
  [ rdfs:label "Votes for B";
    poc:isRequired true;
    poc:fieldType xsd:integer ],
  [ rdfs:label "Votes for C";
    poc:isRequired true;
    poc:fieldType xsd:integer ].
```

Figure 4.3. Definition of *VerifierReport* in *Spec<sub>T3</sub>*, serialized in Turtle.

There are three *DataInstances* defined in *Spec<sub>T3</sub>*: *UnverifiedPhotos* (*UP*), *CandidateReports* (*CR*) and *VerifiedReports* (*VR*). Definition of these *DataInstances* in *Spec<sub>T3</sub>* is demonstrated in Figure 4.4. *UP* is a *List* that contains the *Photos* that are not verified by the community yet. *CR* is a *List* that contains the *VerifierReports* that are provided by *Verifiers*. *UP* and *CR* has *constraints* that restrict the access of these resources for any user. *VR* is a *List* that contains the *VerifierReports* which are regarded as verified, as the information they encapsulate has been provided by three independent *Verifiers*.

```

_:UnverifiedPhotos a poc:List;
  rdfs:label "Unverified Photos"@en;
  poc:constraint [poc:allowIf "false"^^poc:Expression].
_:CandidateReports a poc:List;
  rdfs:label "Candidate Reports"@en;
  poc:constraint [poc:allowIf "false"^^poc:Expression].
_:VerifiedReports a poc:List;
  rdfs:label "Verified Reports"@en.

```

Figure 4.4. Definition of the *DataInstances* in  $Spec_{T3}$ , serialized in Turtle.

There are two *Workflows* defined within  $Spec_{T3}$ : *PhotoSubmission* and *ReportVerification*.

*PhotoSubmission* represents the photograph uploading operation to be performed by *Observers*. The definition of *PhotoSubmission* within  $Spec_{T3}$  is demonstrated in Figure 4.6. A visual representation of *PhotoSubmission* is depicted in Figure 4.5. *PhotoSubmission* has a *constraint* that specifies that it can only be invoked by *Observers*. *PhotoSubmission* encapsulates two *Steps*:  $S_0$  and  $S_1$ .

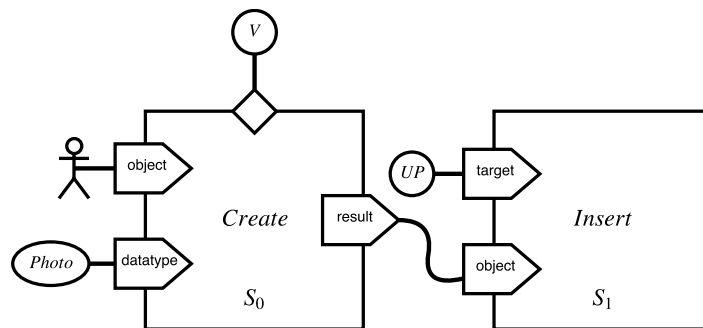


Figure 4.5. A visual representation of *PhotoSubmission*.

$S_0$  involves the creation of an *Image* with a *datatype* of *Photo*. The *task* of  $S_0$  is *Create*, indicating that it involves the creation of a *DataInstance*.  $S_0$  has a

*viewTemplate*, which is identified by *V* in Figure 4.5.  $S_0$  is the *targetStep* of two *Pipes*. The first is a *DirectPipe* whose *targetPort* is "datatype" and *sourceValue* is *Photo*. This *Pipe* indicates that the *DataInstances* to be created are of datatype *Photo*. The second *Pipe* is a *HumanPipe* with a *targetPort* of "object". This *Pipe* indicates that the *DataInstances* to be created are to be provided by human users.  $S_0$  has a *constraint* indicating that it can only be performed by a user who invoked the *Workflow*.

$S_1$  involves the insertion of the *Image* created by  $S_0$  to *UP*. The *task* of  $S_1$  is *Insert*, indicating that it involves the insertion of an item to a *List*. It is the *targetStep* of two *Pipes*. The first is a *DirectPipe* whose *targetPort* is "target" and *sourceValue* is *UP*. This *Pipe* indicates that the *List* the item is to be added is *UP*. The second *Pipe* is a *PortPipe* whose *sourcePort* is "result" and *targetPort* is "object". This *Pipe* indicates that the item to be added is an output of  $S_0$ , and is advertised by its "result" port.

*ReportVerification* is a relatively complex *Workflow*, representing the form filling operation to be performed by *Verifiers*, and the verification mechanism it entails. A visual representation of *ReportVerification* is depicted in Figure 4.7. *Expressions* and *ViewTemplates* involved in this *Workflow* are represented with enumerated labels for brevity, and they represent the following values.

$E_0$  : "@list.size > 0"^^poc:Expression

$E_1$  : "@list.size - 1"^^poc:Expression

$E_2$  : "@report == @item"^^poc:Expression

$E_3$  : "@list.size == 3"^^poc:Expression

$V_0$  : "<p>Please enter the information on the displayed ballot report into the form.</p>

<img src={#@photo.locator#}>"^^poc:ViewTemplate

$V_1$  : "<p>No unverified report at the moment.

Thank you!</p>"^^poc:ViewTemplate

```

_:PhotoSubmission a poc:Workflow;
  rdfs:label "PhotoSubmission"@en;
  dcterms:title "Submit Photo"@en;
  poc:step _:S0, _:S1;
  poc:constraint [poc:allowedRole _:Observer];
  poc:pipe [a poc:HumanPipe; poc:targetStep _:S0; poc:targetPort
    "object"],
    [a poc:DirectPipe; poc:targetStep _:S0; poc:targetPort "datatype";
      poc:sourceValue _:Photo],
    [a poc:PortPipe; poc:sourceStep _:S0 ; poc:sourcePort "result";
      poc:targetStep _:S1 ; poc:targetPort "object"],
    [a poc:DirectPipe; poc:targetStep _:S1 ; poc:targetPort "target";
      poc:sourceValue _:UnverifiedPhotos; ].

_:S0 a poc:Step;
  poc:constraint [poc:allowIf "@user == @invoker"^^poc:Expression];
  poc:viewTemplate "<p>Please upload a photograph of a ballot
    report.</p>";
  dcterms:description "Submit a photograph of a ballot report."@en;
  poc:task poc:Create;
  poc:inputPort [rdfs:label "object"], [ rdfs:label "datatype"];
  poc:outputPort [ rdfs:label "result"].

_:S1 a poc:Step;
  poc:task poc:Insert;
  poc:inputPort [rdfs:label "object"], [ rdfs:label "target"];
  poc:outputPort [rdfs:label "result"].

```

Figure 4.6. Definition of *PhotoSubmission* in *Spec<sub>T3</sub>*, serialized in Turtle.

*ReportVerification* has a *constraint* that specifies that it can only be invoked by *Verifiers*. *ReportVerification* encapsulates ten *Steps*:  $S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}$  and  $S_{11}$ . The functionality of each *Step* and their relations are as follows.

- $S_2$  returns a boolean value indicating whether  $UP$  is empty. If it is not empty,  $S_3$  is invoked, otherwise,  $S_{11}$ .
- $S_3$  generates a random integer between zero and the size of  $UP$  minus one.
- $S_4$  outputs the item of  $UP$  whose index is the number outputted by  $S_3$ . This item is expected to be a *Photo*.
- $S_5$  prompts the user with the *Photo* outputted by  $S_4$ , with a message asking the user to enter data.  $V_0$  is compiled to generate the message to be displayed to the user. The entered data is outputted as a *VerifierReport*.
- $S_6$  inserts the *VerifierReport* generated by  $S_5$  to  $CR$ , and outputs  $CR$ .
- $S_7$  outputs a *List* containing the items of  $CR$  that have the same *fieldValues* as the *VerifierReport* outputted by  $S_5$ .  $S_7$  uses  $E_2$  as the filtering condition.
- $S_8$  checks whether the *List* outputted by  $S_7$  contains exactly three items. If so,  $S_9$  and  $S_{10}$  are invoked.
- $S_9$  inserts the *VerifierReport* outputted by  $S_5$ .
- $S_{10}$  removes the *Photo* outputted by  $S_4$  from  $UP$ , since it is now verified by the system.
- $S_{11}$  informs the user that there is no unverified photographs present.

#### 4.1.2. Execution of a T3 POC Application

Here, we examine the runtime behaviour of a POC Application that conforms  $Spec_{T3}$ , which is the POC Specification described in Section 4.1.1. This POC Application will be referred as  $App_{T3}$ , and assumed to have a POC Server that is identified by the URI `http://example-t3.org`. This section represents a scenario involving multiple users, and discuss the execution behaviour of  $App_{T3}$  and the data interchange between the POC Server of  $App_{T3}$  and its clients.

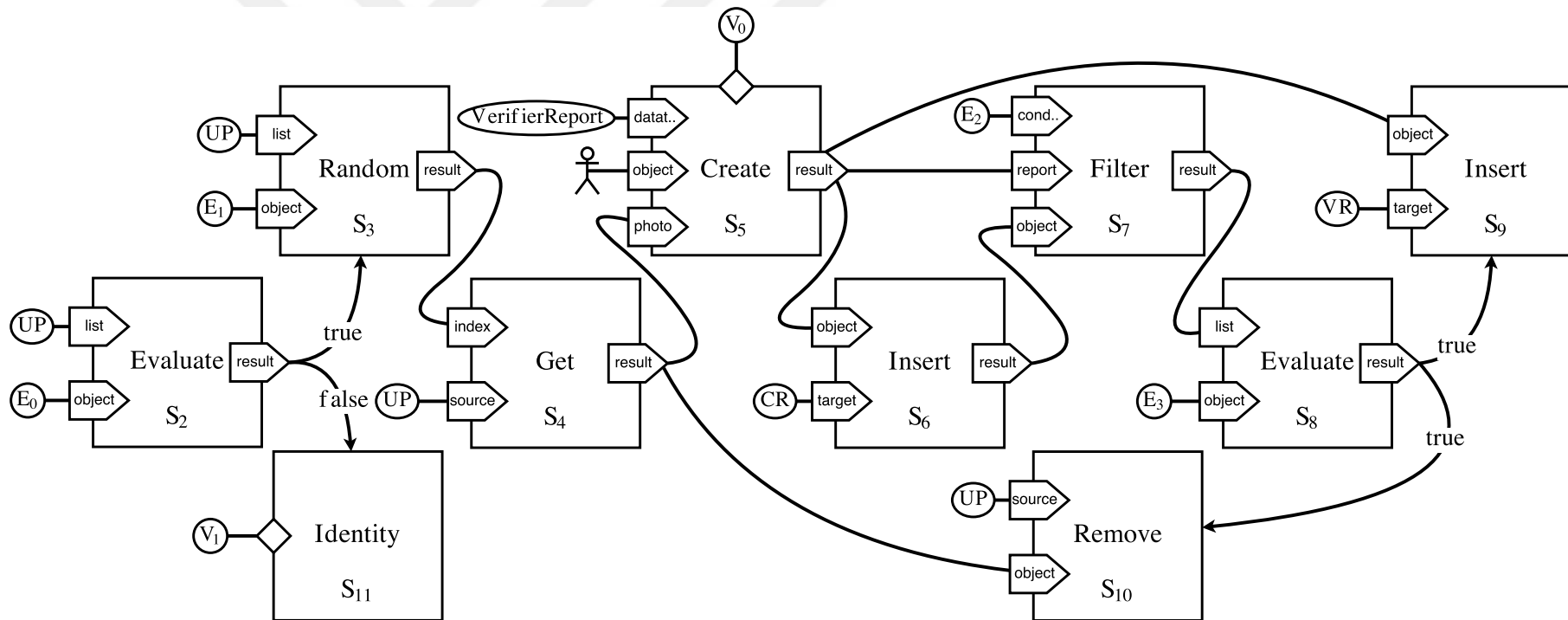


Figure 4.7. A visual representation of *ReportVerification*.

The scenario to be discussed in this section involves numerous HTTP transactions. HTTP headers interchanged within these transactions are mostly omitted, unless they play an important role in the scenario.

Three users are involved in this scenario:  $U_0$ ,  $U_1$  and  $U_2$ .  $U_0$  has a *UserAccount* that is associated with *Verifier* role, and the *UserAccount* of  $U_1$  is associated with *Observer* role.  $U_2$  does not have a *UserAccount*. Since POC Model does not impose a specific authentication mechanism, this scenario omits how the users are authenticated.

The scenario is discussed in four consecutive sections, each involving one of the users.

4.1.2.1. Verification Attempt in the Absence of Photographs. In the first part of our scenario,  $U_0$  attempts to verify reports.

$U_0$  uses a POC Client to retrieve a representation of the POC Server of  $App_{T3}$ , submitting the HTTP request demonstrated by Figure 4.8.

```
GET / HTTP/1.1
Host: example-t3.org
Accept: text/turtle
```

Figure 4.8. An HTTP request message on the POC Server of  $App_{T3}$ .

The received response is depicted by Figure 4.9, which indicates that the requested resource is POC Server and advertises available *Workflows* for  $U_0$  through the relative URI `<workflows>`.

$U_0$  triggers an HTTP request to the advertised relative URI to retrieve the available *Workflows*, which is depicted in Figure 4.10.

```

HTTP/1.1 200 OK
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD

<http://example-t3.org> a poc-ldp:POC_Server;
  poc-ldp:dataInstances <http://example-t3.org/data_instances/>;
  poc-ldp:datatypes <http://example-t3.org/datatypes/>;
  poc-ldp:workflows <http://example-t3.org/workflows/> .

```

Figure 4.9. An HTTP response message advertising a representation of the POC Server of  $App_{T3}$ .

```

GET /workflows/ HTTP/1.1
Host: example-t3.org
Accept: text/turtle

```

Figure 4.10. An HTTP request for the retrieval of the *Workflows* available for a user of  $App_{T3}$ .

The response message is depicted by Figure 4.11, which indicates that the requested resource is a LDP Basic Container, with a single element identified by the relative URI `<report_verification>`. This element is a *Workflow* and has "Verify Report" as its *title*, which cues  $U_0$  on the functionality of the *Workflow*.

In order to invoke the advertised *Workflow*,  $U_0$  submits an HTTP POST request to its URI, as depicted in Figure 4.12.

The corresponding HTTP response message is depicted by Figure 4.13. It advertises that a resource is created as a result of the transaction, which is identified by the



```

HTTP/1.1 200 OK
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD

<> a ldp:BasicContainer;
  ldp:contains <report_verification>.

<report_verification> a poc:Workflow;
  dcterms:title "Verify Report"@en.

```

Figure 4.11. An HTTP response message advertising the *Workflows* available for a user of  $App_{T3}$ .

```

POST /workflows/report_verification HTTP/1.1
Host: example-t3.org
Accept: text/turtle

```

Figure 4.12. An HTTP request message for the invocation of *ReportVerification* by  $U_0$ .

Location header. A representation of the created resource is included in the message body. The created resource is a *WorkflowInstance* whose *datatype* is *ReportVerification*. The *status* of this *WorkflowInstance* is "completed", indicating that its execution is finished. It identifies a single *StepInstance* available for  $U_0$  to perform. The representation of this *StepInstance* is included in the response as well. This *StepInstance* has a *view* to be displayed, informing  $U_0$  that there is no unverified report.

4.1.2.2. Photograph Submission. In the second section of the scenario,  $U_1$  adds a photograph of a ballot report to be verified.

```

HTTP/1.1 201 Created
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD
Location: http://example.com/t3/workflows/report_verification/rv0

<> a poc:WorkflowInstance;
  dcterms:created "2015-12-03T23:48:20Z"^^xsd:dateTime;
  dcterms:creator <http://example-t3.org/user_accounts/U0>;
  poc:status "completed";
  poc:datatype <http://example.org/workflows/report_verification>;
  poc-ldp:step_instances </step_instances> .
</step_instances> a ldp:BasicContainer;
  ldp:contains </step_instances/inform>.

</step_instances/inform> a poc:StepInstance;
  poc:datatype
    <http://example-t3.org/workflows/report_verification/steps/s11>;
  dcterms:created "2015-12-03T23:48:20Z"^^xsd:dateTime;
  dcterms:creator <http://example-t3.org/users/U0>;
  poc:status "completed";
  poc:view "<p>No unverified report at the moment. Thank you!</p>".

```

Figure 4.13. The HTTP response message for the invocation of *ReportVerification* by  $U_0$ .

$U_1$  uses a POC Client to trigger an HTTP transaction to receive a representation of the POC Server of  $App_{T3}$ , which is identical to the transaction depicted by Figure 4.8 and 4.9.  $U_1$  requests the available *Workflows*, triggering an HTTP request as depicted by Figure 4.14.

```

GET /workflows/ HTTP/1.1
Host: example-t3.org
Accept: text/turtle

```

Figure 4.14. An HTTP request for retrieving the available *Workflows* for  $U_1$ .

The POC Server answers with an HTTP response message as demonstrated by Figure 4.15. The response message advertises that the represented resource is a LDP Basic Container, which contains a single element identified by the relative URI `<photo_submission>`. This element is a *Workflow* and has "Submit Photo" as its *title*, which cues  $U_1$  on the functionality of the *Workflow*.

```

HTTP/1.1 200 OK
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD

<> a ldp:BasicContainer;
  ldp:contains <photo_submission>.

<photo_submission> a poc:Workflow;
  dcterms:title "Submit Photo"@en.

```

Figure 4.15. An HTTP response message advertising the available *Workflows* for  $U_1$ .

In order to invoke the *Workflow*,  $U_1$  submits an HTTP POST request that is demonstrated by Figure 4.16.

The corresponding HTTP response is depicted by Figure 4.17. The response advertises that a resource is created as a result of this transaction, which is identified by the Location header. A representation of the created resource is included in the

```

POST /workflows/photo_submission HTTP/1.1
Host: example-t3.org
Accept: text/turtle

```

Figure 4.16. An HTTP request for the invocation of *PhotoSubmission* by  $U_1$ .

response body. The created resource is a *WorkflowInstance* whose *datatype* is *PhotoSubmission*. The *status* of this *WorkflowInstance* is "ongoing", indicating that it is invoked and it has *StepInstances* that are not performed. This *WorkflowInstance* has a single *StepInstance* available for  $U_1$  to perform. The representation of this *StepInstance* is included in the response as well.

The *StepInstance* has a "ready" *status*, indicating that it is available to be performed. It also has a *view* to be displayed, informing  $U_1$  that submission of an image is expected. It is identified to be an instance of  $S_0$ , which is discussed in Section 4.1.1. Lastly, the *StepInstance* has a *input* declaration for the *inputPort* labeled "datatype". The declared value is *Photo*, which is also discussed in Section 4.1.1. This cues the used POC Client to prompt  $U_1$  with a form enabling the user to provide an image conforming the constraints defined by *Photo*. The exact means how this image is to be provided by the human user is an implementation detail to be handled by the used POC Client. It can accept a URL of an image hosted on the Web, or automatically the provided image to a hosting service. The response also includes representations of  $S_0$  and *Photo*, saving POC Client from sending additional requests to retrieve their representations.

$U_1$  provides an image with the URL `http://example.org/report.jpg` through the used POC Client, which triggers the HTTP request depicted by Figure 4.18. The submitted request is a HTTP PATCH request, indicating that the request is intended to modify the target resource. A "Content-Type" header with value "text/ldpatch" is provided by the request, indicating that the request message conforms the Linked

```

HTTP/1.1 201 Created
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD
Location: http://example.com/t3/workflows/photo_submission/ps0

<> a poc:WorkflowInstance; poc:status "ongoing";
  poc:datatype <http://example.org/workflows/photo_submission>;
  dcterms:created "2015-12-03T23:52:41Z"^^xsd:dateTime;
  dcterms:creator <http://example-t3.org/user_accounts/u1>;
  poc-ldp:step_instances </step_instances> .
</step_instances> a ldp:BasicContainer;
  ldp:contains </step_instances/submit_photo>.
</step_instances/submit_photo> a poc:StepInstance;
  poc:datatype
    <http://example-t3.org/workflows/photo_submission/steps/s0>;
  dcterms:created "2015-12-03T23:52:41Z"^^xsd:dateTime;
  dcterms:creator <http://example-t3.org/users/u1>;
  poc:status "ready";
  poc:view "<p>Please upload a photograph of a ballot report.</p>";
  poc:input [rdfs:label "datatype"; poc:value
    <http://example-t3.org/datatypes/photo>].
<http://example-t3.org/workflows/photo_submission/steps/s0> a poc:Step;
  poc:inputPort [rdfs:label "datatype"], [ rdfs:label "object"];
  poc:task poc:Create .

<http://example-t3.org/datatypes/photo> a poc:DerivedDatatype;
  rdfs:label "Photo"@en; poc:baseDatatype poc:ImageType;
  poc:minFrameWidth 400; poc:minFrameHeight 600.

```

Figure 4.17. An HTTP request for the invocation of *PhotoSubmission* by  $U_1$ .

Data Patch Format [39].

```

PATCH /workflows/photo_submission/ps0/step_instances/submit_photo
  HTTP/1.1
Host: example-t3.org
Accept: text/turtle
Content-Type: text/ldpatch

Add { <> poc:input [
  rdfs:label "object"
  poc:value [ a poc:Image;
    poc:datatype <http://example-t3.org/datatypes/photo>;
    ma:locator "http://example.org/report.jpg"^^xsd:anyURI
  ].
}.

```

Figure 4.18. An HTTP request message, submitted by  $U_1$  for performing  $S_0$ .

The corresponding HTTP response is depicted by Figure 4.19. The response carries no message but has status code 200, indicating the *StepInstance* was performed successfully.

```

HTTP/1.1 200 OK
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD

```

Figure 4.19. The HTTP response message after the successful performance of  $S_0$  by  $U_1$ .

4.1.2.3. Verification of Ballot Reports. In the third part of the scenario,  $U_0$  attempts to verify ballot reports again.

$U_0$  uses a POC Client to retrieve the description of  $App_{T3}$ , triggering a HTTP transaction which is identical to the one depicted by Figure 4.8 and 4.9.  $U_0$ , then, requests the available *Workflows*, triggering the a HTTP transaction which is identical to the one depicted by Figure 4.10 and 4.11.  $U_0$  invokes the advertised *Workflow* through the HTTP request depicted by Figure 4.20.

```
POST /workflows/report_verification HTTP/1.1
Host: example-t3.org
Accept: text/turtle
```

Figure 4.20. The HTTP request by  $U_0$ , invoking *ReportVerification* for the second time.

The corresponding HTTP response message is depicted by Figure 4.11. The response advertises that a resource is created as a result of this transaction, which is identified by the Location header. A representation of the created resource is included in the response body. The created resource is a *WorkflowInstance* whose *datatype* is *ReportVerification*. The *status* of this *WorkflowInstance* is "ongoing", indicating that it is invoked and it has *StepInstances* that are not performed. This *WorkflowInstance* has a single *StepInstance* available for  $U_0$  to perform. The representation of this *StepInstance* is included in the response as well.

The *StepInstance* has a "ready" *status*, indicating that it is available to be performed. The *StepInstance* is identified to be an instance of  $S_5$ , which is discussed in Section 4.1.1. It has a *view* to be displayed, instructing  $U_0$  to fill the web form with the information depicted in the displayed image. Notice that, the image URL submitted by  $U_1$  is inserted into the *ViewTemplate*  $V_0$  to yield the provided *view*. Lastly, the *StepInstance* has a *input* declaration for the *inputPort* labeled "datatype". The

declared value is *VerifierReport*. This cues the used POC Client to prompt  $U_0$  with a form that asks for individual *DataFields* of *VerifierReport*. The response also includes representations of  $S_5$  and *VerifierReport*, saving POC Client from sending additional requests to retrieve their representations. For the sake of brevity, the representation of *VerifierReport* is trimmed in Figure 4.21. A complete representation of *VerifierReport* can be found in Figure 4.3.

$U_0$  fills the provided form and submits, triggering the HTTP request depicted in Figure 4.22.

The POC Server answers with the HTTP response message depicted by Figure 4.23. The response carries no message but has status code 200, indicating the *StepInstance* was performed successfully.

4.1.2.4. Accessing Verified Reports. In the last part of our scenario,  $U_2$  attempts to access the verified reports by *App<sub>T3</sub>*. In this section, we assume that two other *Verifiers* have verified the *Photo* submitted by  $U_1$  in the second part of the scenario, providing the same information  $U_0$  provided in the third part.

$U_2$  uses a POC Client to retrieve the description of the POC Server of *App<sub>T3</sub>*, triggering a HTTP transaction which is identical to the one depicted by Figure 4.8 and 4.9.  $U_2$ , then, requests the available *DataInstances*, submitting the HTTP request depicted by Figure 4.24.

The POC Server answers with an HTTP response message as depicted by Figure 4.25. The response exhibits a LDP Basic Container, which contains a single element identified by the relative URI `</verified_reports>`.

$U_2$  requests the representation of the accessible data resource, triggering the HTTP request demonstrated by Figure 4.26.



```

HTTP/1.1 201 Created
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD
Location: http://example.com/t3/workflows/report_verification/rv1

<> a poc:WorkflowInstance;
  dcterms:created "2015-12-03T23:56:26Z"^^xsd:dateTime;
  dcterms:creator <http://example-t3.org/user_accounts/u0>;
  poc:status "ongoing";
  poc:datatype <http://example.org/workflows/report_verification>;
  poc-ldp:step_instances </step_instances> .
</step_instances> a ldp:BasicContainer;
  ldp:contains </step_instances/fill_form>.
</step_instances/fill_form> a poc:StepInstance;
  poc:datatype
    <http://example-t3.org/workflows/report_verification/steps/s5>;
  dcterms:created "2015-12-03T23:56:26Z"^^xsd:dateTime;
  dcterms:creator <http://example-t3.org/users/u0>;
  poc:status "ready";
  poc:view "<p>Please enter the information depicted in the photograph
    into the form.</p> <img src='http://example.org/report.jpg'>";
  poc:input [rdfs:label "datatype"; poc:value
    <http://example-t3.org/datatypes/VerifierReport>].
<http://example-t3.org/workflows/PhotoSubmission/steps/s5> a poc:Step;
  poc:inputPort [rdfs:label "datatype"], [ rdfs:label "object"];
  poc:task poc:Create .
<http://example-t3.org/workflows/datatypes/VerifierReport> a ...

```

Figure 4.21. The HTTP response message received after a successful invocation of *ReportVerification*.

```

PATCH /workflows/photo_submission/ps0/step_instances/submit_photo
      HTTP/1.1
Host: example-t3.org
Accept: text/turtle
Content-Type: text/ldpatch

Add { <> poc:input [
  rdfs:label "object"
  poc:value [ a poc:CompositeDataInstance;
    poc:datatype <http://example-t3.org/datatypes/verifier_report>;
    fieldValue [ rdfs:label "ID"; poc:literalValue 4124 ],
                [ rdfs:label "Votes for A"; poc:literalValue 113 ],
                [ rdfs:label "Votes for B"; poc:literalValue 54 ],
                [ rdfs:label "Votes for C"; poc:literalValue 12 ]
            ]
  ].
}.

```

Figure 4.22. The HTTP request message submitted by  $U_0$  for the performance of  $S_5$ .

```

HTTP/1.1 200 OK
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD

```

Figure 4.23. The HTTP response message received after a successful performance of  $S_5$  by  $U_0$ .

```

GET /data_instances/ HTTP/1.1
Host: example-t3.org
Accept: text/turtle

```

Figure 4.24. The HTTP request message for retrieving the *DataInstances* accessible by  $U_2$ .

```

HTTP/1.1 200 OK
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD
<> a ldp:BasicContainer;
  ldp:contains </verified_reports>.

```

Figure 4.25. The HTTP response advertising the *DataInstances* accessible by  $U_2$ .

```

GET /data_instances/verified_reports HTTP/1.1
Host: example-t3.org
Accept: text/turtle

```

Figure 4.26. The HTTP request message retrieving a representation of *VerifiedReports*.

The POC Server answers with the HTTP response message depicted by Figure 4.27. The response message contains a representation of *VR* which is defined in *Spec<sub>T3</sub>*. *VR* contains a single item. The representation of this item is also included in the response message, which is a *CompositeDataInstance*. This *CompositeDataInstance* has *VerifierReport* as its *datatype* and possesses *fieldValues* that contain the information submitted by  $U_0$  in the third part of the scenario.

```

HTTP/1.1 200 OK
Content-Type: text/turtle
Allow: GET,OPTIONS,HEAD

<> a poc:List;
  poc:items ( <http://example-t3.org/datatypes/verifier_report/vr2> ).

<http://example-t3.org/datatypes/verifier_report/VR2> a
  poc:CompositeDataInstance;
  poc:datatype <http://example-t3.org/datatypes/verifier_report>;
  dataField [ rdfs:label "ID"; poc:literalValue 4124 ],
             [ rdfs:label "Votes for A"; poc:literalValue 113 ],
             [ rdfs:label "Votes for B"; poc:literalValue 54 ],
             [ rdfs:label "Votes for C"; poc:literalValue 12 ].

```

Figure 4.27. The HTTP response message advertising a representation of *VerifiedReports*.

### 4.1.3. An Assessment of T3 POC Application

In Section 4.1.1, a POC Specification for T3,  $Spec_{T3}$ , is demonstrated. In Section 4.1.2, the runtime behaviour of a POC Application conforming  $Spec_{T3}$ ,  $App_{T3}$ , is examined. In this section, we recite the requirements previously identified for T3, and provide an assessment of how well  $App_{T3}$  addresses them.

The first requirement identified for T3, Requirement (i), indicates that *Observers* should be able to submit photographs. *PhotoSubmission* represents this action, which is a *Workflow* defined in  $Spec_{T3}$ . It has a *constraint* specifying that it is allowed for *Observers*. Section 4.1.2.2 demonstrates a scenario describing the execution behaviour of *PhotoSubmission*.

Requirement (ii) indicates that people who are not *Observers* should be restricted from submitting photographs. *PhotoSubmission* has a *constraint* whose *allowedRole* is *Observer*, indicating that possession of *Observer* is required for invoking this *Workflow*. The scenario discussed in Section 4.1.2.1 reveals how a user who is not an *Observer* is not provided with the information of *PhotoSubmission*.

Requirement (iii) indicates that *Verifiers* should be prompted with a random photograph and a web form for capturing the information of ballot reports. *ReportVerification* encapsulates this action, which is a *Workflow* defined in *Spec<sub>T3</sub>*. It has a *constraint* specifying that it is allowed for *Verifiers*.  $S_5$  is a *step* of *ReportVerification* that represent the operation of prompting a *Verifier* with a photograph and capturing the report information. The scenario discussed in Section 4.1.2.3 explains this mechanism.

Requirement (iv) indicates that if there is no photographs to be verified, the *Verifiers* should be informed so.  $S_{11}$  is a *step* of *ReportVerification* that represent the action of informing a *Verifier* if there is no *Photo* in *UP*. The scenario discussed in Section 4.1.2.1 demonstrates how  $S_{11}$  is enacted.

Requirement (v) indicates that the verified reports should be available for any user. The *DataInstances* defined within a POC Specification are available for any user unless the access is restricted by a *Constraint*. *VR*, which is the *List* containing verified reports, has no *constraint*, and is therefore available for any user. Section 4.1.2.4 demonstrates a scenario involving an unauthenticated user to access the content of *VR*.

We conclude that POC Model has enough expressive power to construct an application that address the requirements we identified for T3.

## 4.2. An Application Generating Prototype

In this section, we briefly describe the prototype developed within the scope of our work.

The developed prototype is a web based interpreter for POC Specifications. More specifically, it is a Ruby on Rails applicaiton that utilize a Fuseki RDF database and a Redis in-memory cache store, which are linked together by a Docker container. It is hosted on a virtual machine on Azure, and deployed through a continuous deployment mechanism that utilize CircleCI and Github.

This prototype is developed solely as a sanity check for the author, and is not used for a systematic evaluation of this study. It helped us verify that the *Workflows* encapsulating multiple steps can be automatically performed, given that they do not implement loops. The prototype utilizes a Ruby implementation of Tarjan’s strongly connected components algorithm [55] to obtain a topological sorting of the *Steps* within a *Workflow*. This sorting is used by selecting which *Step* should be invoked next.

The prototype provides a web page that enable submission of POC Specifications in Turtle, which is depicted in Figure 4.28.

---

## **Submit a file containing a POC specification serialized in Turtle format!**

| No file selected.

Figure 4.28. A screenshot of the submission page.

When a valid POC Specification is submitted, the prototype displays an endpoint for a POC Application with limited capability, which reflects a certain set of structures contained within the POC Specification. The endpoint displaying page is depicted in Figure 4.29.

The advertised endpoints only support Turtle. The advertised endpoint can be consumed with numerous tools. Figure 4.29 demonstrates a screenshot of its con-

The generated API for the provided file is accessible through **http://137.116.245.97:3000/api** and supports only 'text/turtle' content-type.

Figure 4.29. A screenshot of the endpoint displaying page.

sumption with HttpRequester [56] which is a browser add-on enabling transmission of custom HTTP requests.

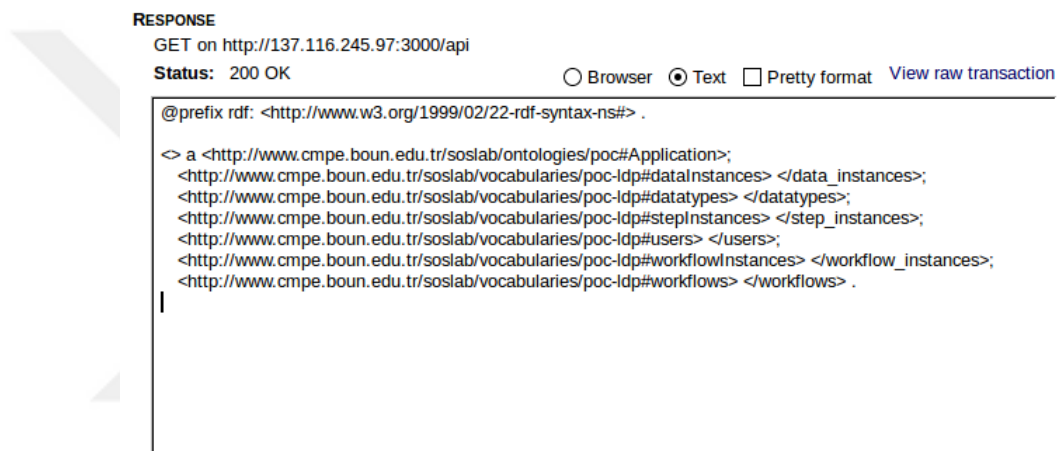


Figure 4.30. A screenshot demonstrating the consumption of an endpoint advertised by the prototype.

## 5. RELATED WORK

In this chapter, we represent some related literature and tools, and discuss how they relate with our work.

### 5.1. Related Work Utilizing Semantic Web Technologies in the Context of Online Communities

The Semantic Web technologies have been utilized by many studies to express the online communities and to examine their dynamics.

Some Web ontologies were developed to represent online community related data, e.g. FOAF [57] vocabulary specifies concepts related to human collaboration, friendship and association, SIOC [44, 45] ontology describes discussion forum and post related concepts in online communities, and SCOT [58] models the tagging activities.

Mika 2005 [59] proposed using tripartite graphs to represent social networks of actors, concepts and resources. Tripartite graphs provided a richer representation of the online community data, than raw graphs. However, instead of manipulating a tripartite graphs directly, Mika reduced it to three bipartite graphs in advance. Using this approach, Mika 2005 [60] constructed a social network of Semantic Web community from the FOAF profiles, academic publications and e-mails crawled from the web. He used JUNG, a Java Graph API, to evaluate Social Network Analysis (SNA) metrics, and provided an interface to visualize the network and the computed metrics.

Some subsequent studies [61–63] revealed that the Semantic Web technologies are powerful enough to make meaningful manipulations over this rich data directly, including computation of SNA measures.

San Martín *et al.* 2009 [61] proposed a data model for social networks based on RDF, and a query and transformation language based on SPARQL 1.0. The proposed



data model represented the relations in the social networks as concepts, enabling one to define n-ary relations. The language they developed provided the ability to query for some SNA measures requiring aggregation, however cannot be used to compute SNA metrics depending on paths, such as betweenness centrality. [61] set a good example for representing, querying and transforming the data of social networks, yet, their methodology is out-of-date with the subsequent release of SPARQL 1.1, which is inherently capable of aggregation.

Erétéo *et al.* 2009 [62] developed an ontology called SemSNA [64] that models a set of SNA notions. SemSNA is meant to be combined with a domain ontology, modeling a social network of any domain. SemSNA aims to provide means to abstract social network constructs from the domain ontologies, and enrich the concepts of domain ontologies with annotations of Social Network Analysis indices. [62] also proposes a procedure to compute and store the SNA measures. The developed procedure makes use of a SPARQL search engine, CORESE, which extends SPARQL with aggregation and path retrieval features. They used CORESE to generate RDF dumps from relational databases and performed CONSTRUCT queries of SPARQL to create instances of SNA concepts and associate them with the actor instances. The procedure enables one to update the SNA related constructs and indices as the network changes, without calculating everything from the beginning. It is noteworthy that while the procedure proposed in [62] is capable of computing SNA measures requiring aggregation or path computation, it also defines new SNA measures exploiting the rich representation of the network data.

Angeletou *et al.* 2011 [63] approached the online communities from a different angle. Rather than SNA, [63] focused on the behavioral roles exhibited by the users, with an aim to determine and forecast the health of the online communities. The authors used a predefined set of behavioral roles, and specified different activity patterns required for each role. They also defined a behavior ontology [65] that models the online community users, their interactions and behavioral roles. The study proposes a method that tests each user against the activity patterns to identify the behavioral roles they hold, and produce a role composition of the online community. The pro-

posed method adjusts the numerical boundaries for the activity patterns dynamically, updating the role extraction queries via SPIN framework extending SPARQL. This enables the role composition extraction process to be applicable to online communities of different sizes, and thus repeatable on an evolving online community in different periods.

ActivityStreams 2.0 [66] is specification that is currently being developed by W3C, aiming to provide a JSON-based syntax for expressing various activities in a machine-interpretable manner. It is intended to provide a standard format for the interchange of social Web application data, in terms of activities performed through those applications. Activity Vocabulary [67] is a vocabulary of the classes and properties utilized by ActivityStreams 2.0 to specify activities.

Many of the mentioned work involve Web ontologies that are used either to specify the actions performed within online communities, or to specify some meta-data related with such actions. None of these ontologies, however, aim to specify the underlying application behavior that facilitate the actions on online communities. The most fundamental difference of POC Core Ontology with such ontologies is that, POC Core Ontology models the inner mechanics of online applications as well as the data generated or advertised by them.

## **5.2. Related Work on the Automated Generation of Application Behavior**

There has been numerous efforts towards the automated generation of application behavior. Some of these work modeled the application behavior as pipelines, consisting of predefined tasks put together with data dependencies. Yahoo! Pipes [68] brought the pipes metaphor to the Web, providing a graphical user interface to define pipes that process XML documents (which are mostly in RSS) and generate mash-ups. XProc [69] is an XML pipeline language, recommended by W3C to be used to describe processing operations on XML documents. XProc aims to provide a scalable and reusable method to construct XML processing workflows, representing basic processing operations as small, simple pipelines, which can be combined into larger, complex pipelines.

An XML Pipeline specifies a sequence of operations to be performed on zero or more XML documents. These pipelines accept zero or more XML documents as input and produce zero or more XML documents as output. XML Pipelines are made up of simple steps, which perform atomic operations on XML documents, and auxiliary constructs, i.e., conditionals, iteration, and exception handlers. There are three types of steps in XProc: atomic, compound, multi-container. Atomic steps perform a single operation. A library of atomic steps is provided by XProc. A compound step is one that contains at least one subpipeline. A multi-container step consists of two or more alternate subpipelines, which are either processed depending on a conditional statement, or used to construct a try-catch mechanism. Any XML Pipeline can be used as a step, or subpipeline, in another pipeline. The design of *Workflows* in POC Model is inspired from XML Pipelines in XProc, thereby, there is a strong resemblance between their structures. Both *Workflows* and XML Pipelines consist of steps that represent individual operations with interdependencies. The library of atomic steps in XProc are analogous with the predefined set of *Tasks* in POC Model. An important difference is, ofcourse, that XProc handles automated processing of XML documents, whereas, POC Model targets Web applications that involve asynchronous operations to be performed automatically or with human input.

Le-Phuoc *et al.* [70] proposes Semantic Web Pipes(SWP), a conceptual pipeline framework for semantic data. SWP expresses semantic data mash-ups as pipelines of operations on serialized RDF data. The operations supported by SWP includes SPARQL CONSTRUCT and SELECT queries, operations to load RDF and XML data, split and merge of RDF documents and inference for RDFS and OWL. The pipeline mechanism implemented by SWP is similar with the one presented by XProc for XML documents. The authors implemented a web-based prototype of the proposed system, which provides a graphical interface to be used to construct Web pipes. The developed Web pipes are stored as XML documents, and can be shared and reused.

Kokciyan *et al.* [71] proposes a framework, WeFlow, that generates collaborative human computation applications. The authors modeled an application as a single collaborative workflow, consisting of tasks having control and data interdependencies,

which can be performed by different users. WeFlow follows a three step process for the generation of the applications: specification of the workflow, generation of the application, execution of the generated application. The first step involves providing a specification file in XML. Then, the application behavior is generated according to the provided specification. Lastly, the execution engine runs the application and serves the application workflow as a set of interlinked web views. The authors note that the system does not support concurrency. Another framework to generate applications is Simple Flow by Jara *et al.* [72]. Simple Flow provides uses a single directed graph to model the control flow between the actions performable by the users. The tool provides a simple user interface to be used to design the control flow of the application. Using the interface, the community builder selects performable actions from a predefined set of actions and specifies their dependencies. The constructed action graphs are then interpreted by the tool to create the web views dynamically.

Curbera *et al.* [73] developed Bite, which is an explicit, workflow based composition model for Web applications. Bite represents a business process as a single graph of activities with data dependencies in between, which is stored in an XML document. The model follows the principle of “use implies definition”, and thus the data being transmitted between the activities is not strongly typed. Each business process in Bite is assigned a URL. An HTTP POST request against such a URL initiates an instance of the associated process and returns the URL of the initiated instance. The proposed model supports asynchronous execution of the business logic and multi-party interactions.

## 6. DISCUSSION AND FUTURE WORK

In this chapter, we discuss the current state of our work and its future prospects.

As discussed in Section 2, the Semantic Web technologies need a widespread adoption to reveal their power. This, however, has not been succeeded as anticipated in 2001 [23], although most of the technologies in the Semantic Web stack are available for a considerable time. We think that an important cause of this lack of short term incentives for application developers to bear the nontrivial adoption of these technologies. This issue could be tackled by application building frameworks that provide support for such technologies, without imposing the burden of understanding them to the application developers. We think the approach proposed within POC Model is a promising step in this direction, as it facilitates specification of applications that consume and advertise Linked Data.

In Section 4.1, we concluded that the proposed model is capable of addressing the needs of a real world community. However, it has a number of significant shortcomings, which would hinder its effectiveness in a complex production environment.

An important shortcoming the ambiguity of the behaviour of POC Applications in case of errors. The proposed model does not specify what happens to a *WorkflowInstance* if an intermediate *StepInstance* fails its execution, or simply if the user provides the wrong type of input. This issue can be addressed with a systematic list of error definitions, identifying which kind of error occurs in which case, and how these errors should be handled by a POC Application.

Conceptualization provided by POC Core Ontology have apparent drawbacks as well. The provided domain model does not allow definition of complex *Tasks* that are combinations of the set of primitive *Tasks* provided by the ontology. This hinders reusability of POC Specifications, since *Workflows* are not reusable. This issue can be addressed with introducing a new complex Task type, which resemble *Workflows* but

have input and outputs.

Moreover, *Steps* and *Pipes* of a *Workflow* constructs a directed graph, where each *Step* is represented as a node and each *Pipe* is represented as an edge. Topological sorting of this graph is required for an unambiguous runtime behaviour, which is only possible for acyclic graphs. Therefore, loops cannot be expressed by our current conceptualization.



## 7. CONCLUSION

In this chapter, we provide conclusive comments on the work described within this document.

We proposed a novel ontology-driven model for building community-centric online applicaitons that consume and publish Linked Open Data. We also introduced an novel OWL 2 ontology that models such community-centric applications in terms of community specific information types and workflows. We attempted to assess the expressivity of the proposed model through a real purposeful online community, and identified a number of shortcomings it exhibits.

## REFERENCES

1. Twitter, I., “Twitter”, <http://www.twitter.com/>, accessed at January 2016.
2. Facebook, I., *Facebook*, accessed at January 2016.
3. Howard, P. N. and M. M. Hussain, “The role of digital media”, *Journal of democracy*, Vol. 22, No. 3, pp. 35–48, 2011.
4. Lovejoy, K. and G. D. Saxton, “Information, community, and action: how nonprofit organizations use social media”, *Journal of Computer-Mediated Communication*, Vol. 17, No. 3, pp. 337–353, 2012.
5. Eltantawy, N. and J. B. Wiest, “The Arab spring| Social media in the Egyptian revolution: reconsidering resource mobilization theory”, *International Journal of Communication*, Vol. 5, p. 18, 2011.
6. Dugdale, J., B. Van de Walle and C. Koeppinghoff, “Social media and SMS in the Haiti earthquake”, *Proceedings of the 21st international conference companion on World Wide Web*, pp. 713–714, ACM, 2012.
7. Group, W. O. W., “OWL 2 Web Ontology Language Document Overview (Second Edition)”, <http://www.w3.org/TR/owl2-overview/>, accessed at December 2015.
8. Hitzler, P. and M. Krötzsch, “OWL 2 Web Ontology Language Primer (Second Edition)”, <http://www.w3.org/TR/owl2-primer/>, accessed at December 2015.
9. Berners-Lee, T., “Linked Data”, <http://www.w3.org/DesignIssues/LinkedData.html>, accessed at December 2015.
10. Law, E. and L. v. Ahn, “Human computation”, *Synthesis Lectures on Artificial Intelligence and Machine Learning*, Vol. 5, No. 3, pp. 1–121, 2011.



11. Von Ahn, L., B. Maurer, C. McMillen, D. Abraham and M. Blum, “recaptcha: Human-based character recognition via web security measures”, *Science*, Vol. 321, No. 5895, pp. 1465–1468, 2008.
12. Von Ahn, L., “Games with a purpose”, *Computer*, Vol. 39, No. 6, pp. 92–94, 2006.
13. Amazon.com, I., “Mechanical Turk”, <https://www.mturk.com/mturk/welcome>, accessed at December 2015.
14. Buhrmester, M., T. Kwang and S. D. Gosling, “Amazon’s Mechanical Turk a new source of inexpensive, yet high-quality, data?”, *Perspectives on psychological science*, Vol. 6, No. 1, pp. 3–5, 2011.
15. von Ahn, L., “Duolingo: learn a language for free while helping to translate the web”, *Proceedings of the 2013 international conference on Intelligent user interfaces*, pp. 1–2, ACM, 2013.
16. Hollingsworth, D. and U. Hampshire, “Workflow management coalition the workflow reference model”, *Workflow Management Coalition*, Vol. 68, p. 26, 1993.
17. Fielding, R. T., *Architectural styles and the design of network-based software architectures*, Ph.D. Thesis, University of California, Irvine, 2000.
18. Berners-Lee, T., R. Fielding and L. Masinter, *Uniform resource identifier (URI): Generic syntax*, Tech. rep., 2004.
19. Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, *Hypertext transfer protocol–HTTP/1.1*, Tech. rep., 1999.
20. Fielding, R. T. and R. N. Taylor, “Principled design of the modern Web architecture”, *ACM Transactions on Internet Technology (TOIT)*, Vol. 2, No. 2, pp. 115–150, 2002.
21. Fielding, R. T., “REST APIs must be hypertext-driven”, <http://roy.gbiv.com/>

- untangled/2008/rest-apis-must-be-hypertext-driven, accessed at December 2015.
22. Jaffe, J., “The World Wide Web Consortium”, <https://www.w3.org/>, accessed at December 2015.
  23. Berners-Lee, T., J. Hendler, O. Lassila *et al.*, “The semantic web”, *Scientific american*, Vol. 284, No. 5, pp. 28–37, 2001.
  24. Cyganiak, R. and D. Wood, “RDF 1.1 Concepts and Abstract Syntax”, <http://www.w3.org/TR/rdf11-concepts/>, accessed at December 2015.
  25. Schreiber, G. and Y. Raimond, “RDF 1.1 Primer”, <http://www.w3.org/TR/rdf11-primer/>, accessed at December 2015.
  26. Beckett, D. and B.-L. T., “RDF 1.1 Turtle”, <http://www.w3.org/TR/turtle/>, accessed at December 2015.
  27. Sporny, M. and D. Longley, “JSON-LD 1.0: A JSON-based Serialization for Linked Data”, <http://www.w3.org/TR/json-ld/>, accessed at December 2015.
  28. Gandon, F. and S. G., “RDF 1.1 XML Syntax”, <http://www.w3.org/TR/rdf-syntax-grammar/>, accessed at December 2015.
  29. Group, T. W. S. W., “SPARQL 1.1 Overview”, <http://www.w3.org/TR/sparql11-overview/>, accessed at December 2015.
  30. Brickley, D. and R. V. Ruha, “RDF Schema 1.1”, <http://www.w3.org/TR/rdf-schema/>, accessed at December 2015.
  31. Horrocks, I. and P. F. Patel-Schneider, “SWRL: A Semantic Web Rule Language Combining OWL and RuleML”, <http://www.w3.org/Submission/SWRL/>, accessed at December 2015.

32. Schmachtenberg, M., C. Bizer, A. Jentzsch and R. Cyganiak, “Linking Open Data cloud diagram 2014”, <http://lod-cloud.net/>, accessed at December 2015.
33. Auer, S., C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak and Z. Ives, *Dbpedia: A nucleus for a web of open data*, Springer, 2007.
34. Bizer, C., J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak and S. Hellmann, “DBpedia-A crystallization point for the Web of Data”, *Web Semantics: science, services and agents on the world wide web*, Vol. 7, No. 3, pp. 154–165, 2009.
35. Wikimedia, F., “Wikipedia”, <https://wikipedia.org/>, accessed at December 2015.
36. Freudenberg, M. and D. Kontokostas, “DBpedia Version 2015-04 released”, <http://blog.dbpedia.org/?p=148>, accessed at December 2015.
37. Bergman, M. and F. Giasson, “Upper Mapping and Binding Exchange Layer (UMBEL) Specification”, [http://techwiki.umbel.org/index.php/UMBEL\\_Specification](http://techwiki.umbel.org/index.php/UMBEL_Specification), accessed at December 2015.
38. Speicher, S. and J. Arwe, “Linked Data Platform 1.0”, <http://www.w3.org/TR/ldp/>, accessed at December 2015.
39. Bertails, A. and P. Champin, “Linked Data Patch Format”, <http://www.w3.org/TR/ldpatch/>, accessed at December 2015.
40. Fielding, R. and J. Reshke, “Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing”, <https://tools.ietf.org/html/rfc7230>, accessed at December 2015.
41. Dusseault, L. and J. Snell, “PATCH Method for HTTP”, <https://tools.ietf.org/html/rfc5789>, accessed at December 2015.

42. Weibel, S., “The Dublin Core: a simple content description model for electronic resources”, *Bulletin of the American Society for Information Science and Technology*, Vol. 24, No. 1, pp. 9–11, 1997.
43. Board, D. U., “Dublin Core Vocabulary”, <http://dublincore.org/documents/dcmi-terms/>, accessed at December 2015.
44. Breslin, J. G., A. Harth, U. Bojars and S. Decker, “Towards semantically-interlinked online communities”, *The Semantic Web: Research and Applications*, pp. 500–514, Springer, 2005.
45. Bojārs, U. and J. Breslin, “SIOC Core Ontology Specification”, <http://rdfs.org/sioc/spec/>, accessed at December 2015.
46. Lee, W. and T. Bailer, “Ontology for Media Resources 1.0”, <http://www.w3.org/TR/mediaont-10/>, accessed at December 2015.
47. Peterson, D. and S. Gao, “W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes”, <http://www.w3.org/TR/xmlschema11-2/>, accessed at December 2015.
48. Vatant, B. and M. Wick, “GeoNames Ontology”, <http://www.geonames.org/ontology>, accessed at December 2015.
49. Snell, J., “Prefer Header for HTTP”, <https://tools.ietf.org/html/rfc7240>, accessed at December 2015.
50. Fielding, R. and J. Reshke, “Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content”, <https://tools.ietf.org/html/rfc7231>, accessed at December 2015.
51. Kernighan, B. W., D. M. Ritchie and P. Ekelint, *The C programming language*, Vol. 2, prentice-Hall Englewood Cliffs, 1988.

52. Flanagan, D. and Y. Matsumoto, *The ruby programming language*, " O'Reilly Media, Inc.", 2008.
53. Hickson, A. and R. Berjon, "HTML 5 - A vocabulary and associated APIs for HTML and XHTML", <http://www.w3.org/TR/html5/>, accessed at December 2015.
54. Oy ve Ötesi, F., "An assessment of the Turkish general elections of 2015", <http://oyveotesi.org/7-haziran-2015-genel-secim-sonuc-degerlendirmeleri/>, accessed at December 2015.
55. Tarjan, R., "Depth-first search and linear graph algorithms", *SIAM journal on computing*, Vol. 1, No. 2, pp. 146–160, 1972.
56. Mutdosch, T., "HTTPRequester", <https://addons.mozilla.org/en-US/firefox/addon/httprequester/>, accessed at December 2015.
57. Brickley, D. and L. Miller, "FOAF Vocabulary Specification 0.99", <http://xmlns.com/foaf/spec/>, accessed at December 2015.
58. Kim, H. L. and J. G. Breslin, "Basic Geo (WGS84 lat/long) Vocabulary", <http://rdfs.org/scot/spec/>, accessed at December 2015.
59. Mika, P., "Ontologies are us: A unified model of social networks and semantics", *The Semantic Web–ISWC 2005*, pp. 522–536, Springer, 2005.
60. Mika, P., "Flink: Semantic web technology for the extraction and analysis of social networks", *Web Semantics: Science, Services and Agents on the World Wide Web*, Vol. 3, No. 2, pp. 211–223, 2005.
61. San Martín, M. and C. Gutierrez, "Representing, querying and transforming social networks with RDF/SPARQL", *The Semantic Web: Research and Applications*, pp. 293–307, 2009.

62. Erétéo, G., M. Buffa, F. Gandon and O. Corby, *Analysis of a real online social network using semantic web frameworks*, Springer, 2009.
63. Angeletou, S., M. Rowe and H. Alani, “Modelling and analysis of user behaviour in online communities”, *The Semantic Web–ISWC 2011*, pp. 35–50, Springer, 2011.
64. Erétéo, G., M. Buffa, F. Gandon and O. Corby, “SemSNA Ontology”, <http://ns.inria.fr/semsna/2009/06/21/voc.rdf>, accessed at December 2015.
65. Angeletou, S., M. Rowe and H. Alani, “Behaviour Ontology”, <http://people.kmi.open.ac.uk/miriam/ontology/BehaviourOntology.n3>, accessed at December 2015.
66. Snell, J. M. and E. Prodromou, “Activity Streams 2.0”, <https://www.w3.org/TR/activitystreams-core/>, accessed at December 2015.
67. Snell, J. M. and E. Prodromou, “Activity Vocabulary”, <https://www.w3.org/TR/activitystreams-vocabulary/>, accessed at December 2015.
68. Sadri, P. and E. Ho, “Yahoo! Pipes”, <https://pipes.yahoo.com>, accessed: 2014-05-12.
69. Walsh, N., A. Milowski and H. S. Thompson, “Xproc: An xml pipeline language”, *a conference on XML*, p. 13, 2007.
70. Le-Phuoc, D., A. Polleres, M. Hauswirth, G. Tummarello and C. Morbidoni, “Rapid prototyping of semantic mash-ups through semantic web pipes”, *Proceedings of the 18th international conference on World wide web*, pp. 581–590, ACM, 2009.
71. Kokciyan, N., S. Uskudarli and T. Dinesh, “User generated human computation applications”, *Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Confernece on Social Computing (Social-Com)*, pp. 593–598, IEEE, 2012.

72. Jara, J., F. Daniel, F. Casati and M. Marchese, “From a simple flow to social applications”, *Current Trends in Web Engineering*, pp. 39–50, Springer, 2013.
73. Curbera, F., M. Duftler, R. Khalaf and D. Lovell, *Bite: Workflow composition for the web*, Springer, 2007.

