A HYBRID APPROACH TO DEVELOPING ONTOLOGY-DRIVEN APPLICATIONS: A CASE STUDY IN DESCRIBING RADIO SPECTRUM USAGE POLICIES

Owen Xie

Submitted in Partial Fullfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Approved by:
Dr. Deborah McGuinness, Chair
Dr. James Hendler
Dr. Ana Milanova

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York

[May 2021]
Submitted March 2021

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to first and foremost, thank my advisor, Dr. Deborah McGuinness for her feedback about my research throughout the process, as well as agreeing to chair the committee for my thesis. I would also like to thank Dr. Jim Hendler and Dr. Ana Milanova for their feedback on my work and agreeing to become part of my committee. I am also thankful to Dr. Henrique Santos for his advice about the research process. Finally, I would like to thank Dr. Minor Gordon for his constant support through my research and our discussions about program design and programming languages.

ABSTRACT

The Semantic Web initiative pushes for use of open specifications (e.g. Resource Description Framework (RDF) and Web Ontology Language (OWL)) to describe data and support automated inference from machine-readable logical rules in Ontology-Driven Applications. In such applications, the domain-specific data is often directly parsed from the graph-based data model. However, when the code and the ontology are tightly linked, the flexible nature of ontology development makes code maintenance difficult in the long term. For example, the structure of an ontology often changes to add more semantics or to enable automated reasoners to apply complex business logic to the data (e.g. reasoning with OWL and HermiT). In many cases, the underlying domain data is left unchanged. Regardless, these changes require developers to update their code to accommodate the new structure. When multiple applications directly rely on the graph, changes propagate across all of them, further slowing down application development and increasing load on the developer. Additionally, with OWL-based ontologies, extensions such as meta-modelling introduce further difficulty when extracting information from an ontology, as it needs complex graph traversal code. This encourages the growth of technical debt over time. As such, applications need a simple and stable API that abstracts much of the structure, but keeps the strengths of OWL in reasoning or clarity. To meet this need, we utilize a hybrid approach to integrating an ontology into domain-specific applications, specifically in the domain of computable policies. A hybrid approach is characterized by choosing what to model in domain-specific classes, while delegating the rest to the underlying knowledge graph. Our approach uses aspects of domain-driven design and a backing domain-specific language to capture the essence of a domain with links to the ontology to preserve the strengths of OWL. To analyze this approach, we describe an implementation of a library that models radio spectrum usage policies in the Dynamic Spectrum Access Policy Framework and discuss it's strengths and weaknesses.

1. INTRODUCTION

The Semantic Web focuses on describing data in a well-documented environment. With a common framework such as the Resource Description Framework (RDF) [1], data is serialized or represented in a graph-like structure based on the subject-predicate-object model. The idea is that a subject is described by relationships. Each relationship includes a predicate and a object, where the predicate describes the meaning of the relationship and the object (which can be a primitive variable or another subject) assigns the other value of the relation. In RDF, subjects (also known as "Resources") and objects are vertices, while predicates are represented with edges.

To ensure that the data is well-described, each vertex or edge is labeled with specific identifier, the URI, that either uniquely identifies an subject or associates a well-defined meaning to a relationship. RDF is notable in this case since it is self-describing in a sense: One can use it to describe the meaning of new concepts with an associated URI in the same graph as the data. This component of the graph is commonly known as an ontology. Formally, the ontology defines a conceptual model that describes terminology and concepts used in the data. Altogether, the ontology and data forms the "knowledge graph" [2]. There are several benefits to this model:

- The model encourages good documentation, which promotes sharing and reuse of the data.
- The model is flexible since it does not restrict adding new fields or relationships that describe an subject, as opposed to relational data or even traditional object-oriented (Java-like) models.
- As a language designed for the web, RDF-based knowledge graphs are designed to be accessible from anywhere. With URIs, nodes can represent anything (e.g. XML documents), forming what is known as the "Linked Data Web".

While RDF provides the basic framework for describing objects in data, there are other standardized ontologies that are usually layered on top of RDF to enable more complex representation in the knowledge graph. For example, the Web Ontology Language (OWL) is a Description Logic (a subset of first-order logic) based language that is used to mix business logic into the data, though the use of "classes". As opposed to Java-like classes,

classes in OWL describe necessary and sufficient conditions for a Resource to be an instance of the class. These conditions are described using subclass and equivalence predicates, with subclass fields describing necessary conditions and equivalence fields describing sufficient conditions [3]. As such, OWL is often used to extend software with support for complex reasoning about assertions in the data through the use of inference engines such as HermiT [4] or Pellet [5].

1.1 Ontologies in Software

Since RDF is a specification of knowledge representation, it is used in software for a wide variety of purposes. Notably, RDF is often used in a class of so-called NoSQL databases known as graph databases (e.g. Amazon Neptune, Blazegraph, etc.) for storing large amounts of data. Likewise, for smaller use cases, there are a variety of serialization formats: RDF/XML [6], JSON-LD [7], and Turtle [8]. As mentioned above, inference engines are also used in conjunction with OWL to infer new information from statements in the knowledge graph.

While such applications are generic enough to use the graph structure of the knowledge graph effectively, the same cannot be said of the domain-specific applications that allow users to view or make changes to the graph. Since the end-user should be able to use the application without any knowledge of the underlying knowledge graph, it comes down onto the programmer to be able to translate from a graph view of the data to something that a domain expert can understand. However, interacting with the graph in code is messy and awkward at best [9]. Application data is often directly derived from the knowledge graph, with the program needing to reference the URI of a predicate / property (often using "magic strings", a known anti-pattern) to view the attributes of an subject. This issue is exacerbated when it comes to deeply nested structures: the program needs convoluted code to traverse the graph, often making multiple passes over the same predicate if it is overloaded (when properties are reused for different purposes). Deep traversal is also an issue for the standard query language for RDF, SPARQL [10], often requiring overuse of special features such as property paths and ending up with a query that is not clear to a programmer on what the intended graph structure to query for is.

This also impacts long-term code maintenance. Since data is often shared across programs in multiple languages (e.g. JavaScript in the front-end, Java in the back-end) the

graph traversal code is often repeated across languages. As a result, any changes to the ontology requires repeated changes, slowing down application development. In many cases, the ontology is changed without changing to the actual meaning of the data (e.g. modifying OWL rules to support reasoning) in the context of the application. Dealing with the technical debt from these issues is a heavy burden on the programmer. Thus, this work aims to address these issues and reduce the load on the programmer, enabling them to think about the program in a more domain-specific context, in the spirit of domain-driven design.

1.2 The DSA Policy Framework

Access control models such as attribute-based access control (ABAC) [11] are frequently used to reduce the complexity of controlling access to a resource. A domain this concept has been applied to is radio spectrum usage. Devices such as radios and phones must comply with the many spectrum policies defined by agencies such as the National Telecommunications and Information Administration (NTIA) and Federal Communications Commission (FCC). As the spectrum has become crowded over time, such as with the recent development of 5G, automation of spectrum management has become especially important.

To support automatic policy management, Santos et al. proposed the Dynamic Spectrum Access Policy Framework (DSA Policy Framework), which applies machine-readable radio spectrum usage policies to evaluate spectrum access requests based on attributes of the requester [12]. The policies are represented with a combination of OWL and PROV-O (The PROV Ontology) [13] and are stored in a knowledge graph, which is then used to infer which policies are applicable to an access request. As a management framework, the DSA Policy Framework also provides several interfaces for spectrum managers to explore, edit or create policies. The interfaces require that the policy representation is passed between different tools and across different languages. As a result, this works focuses on how it extends the DSA Policy Framework to support this transfer of data in a cleaner approach. Furthermore, the work shows how this extension enables more ways to view and edit of radio spectrum policies through the use of a domain-specific language (DSL).

1.3 Thesis Overview

This thesis aims to utilize hybrid modelling to model computable policies in ontology driven applications. To determine it's feasibility, we evaluate it in the context of the DSA Policy Framework, which reasons over such policies. The hybrid modelling is used to create a higher level model while still maintaining the benefits of the underlying knowledge graph. We use our evaluation to show that a hybrid approach can handle complex domains and provides a large benefit to the programmer, enabling good development practices and reducing the load on the programmer. We also aim to extend hybrid modelling with the inclusion of a domain-specific language, which is useful for describing data a concise and domain-specific manner.

In the DSA Policy Framework, we implemented a Scala (a Java-like language) library, which we call *DsaModels*, that translates between the higher level model and the underlying knowledge graph. Something to note is that *DsaModels* is already in use in the DSA Policy Framework, where it is used to create user interfaces to view radio spectrum policies and serialize the policies in non RDF-based formats.

The next chapter gives the background for several topics: how the DSA Framework uses OWL, previous approaches to integrating ontologies in software, and the background and motivation for using domain-specific languages. Chapter 3 details the how the concepts for spectrum management is modeled in software, while Chapter 4 discusses the implementation of the library that extends the DSA Framework. Finally, Chapter 5 evaluates the extension to the DSA Framework and Chapter 6 concludes the work and discusses future work.

2. LITERATURE

2.1 Using Ontologies in Software

To solve problems, software needs to be domain-specific somewhere. With regards to development of Ontology Driven Architectures, directly deriving domain-specific application data from the structure of the knowledge graph restricts development [14, 9, 15]. There have mainly been two approaches to address this issue: in ontology or in software.

2.1.1 Ontology Programming Interfaces

Rector et al. introduced the concept of an Ontology Programming Interface, a separate sub-ontology for the application that is expected to remain stable [15]. The underlying ontology is then bound to the sub-ontology using a "Ontology Binding Interface module", which can use either equivalence relations or sub classes to attach fields from the base ontology to the sub-ontology. While this approach has the benefit of encouraging modular design of ontologies, the approach is limited at the identifier level, as there is little known research into automatically mapping nested structures into sub-ontologies. This motivates us to look to a more-software based approach.

2.1.2 Direct, Indirect, and Hybrid Modelling in Software

There are many ontology-oriented programming libraries that allow for development of Linked Data and Semantic Web applications. With regards to integrating OWL with software, Puleston et al. categorized the different methods of working with OWL in Java into direct, indirect, and hybrid approaches [9]. A direct approach models the entire domain in Java, where each Java class and field corresponds to an OWL class, or "model entity". This approach is typically associated with automated Java code generation, such as with the application library OWL2Java[16]. On the other hand, an indirect approach instantiates each "model entity" as an instance of a generic or domain-neutral class and the fields of the class are responsible for identifying it. Examples of APIs that allow for indirect modelling include Apache Jena [17] and OwlReady [18]. Finally, a hybrid approach, as its name suggests, is a combination of the previous approaches. The hybrid approach uses a a small number of Java

classes that form the backbone of the model, while the remaining entities are (indirectly) modelled in the ontology [9]. While there is no standard heuristic for what should be in Java or in the ontology, the core Java classes usually define a domain-specific structure with fields filled by the identifiers in the ontology to fill in the details. In many cases, the indirect model makes up the majority of the domain knowledge.

There are tradeoffs to all approaches. Direct modelling provides a intuitive API for developing domain-specific applications, at the cost of expressiveness. For example, information about an entity is limited to common features, which may leave out data that makes the entity unique. On the other hand, indirect modelling maintains the expressiveness of OWL modelling, but introduces awkward code in domain-specific applications. Hybrid modelling aims to capture the strengths of both approaches, in that the API should be intuitive to use, while providing references to the expressive modelling that an ontologist might introduce. However, there are some limitations to indirect model access, which we discuss in chapter 6.

2.2 Computable Policies

Since the spectrum is shared between many organizations, radio systems, or radio devices, regulatory policies are used to minimize interference. As a policy evaluation tool, the DSA Policy Framework receives requests to transmit over the spectrum and uses the policies to evaluate whether a device (designated as the requester) is allowed to transmit or not. As such, the framework needs to be able to represent a policies in a form that a reasoner can process (what we call "computable policies").

In terms of computable policies, previous approaches include OWL-S [19] and SWSL [20]. More specific applications include attribute-based access control (ABAC) [11], with a notable standard being XACML (eXtensible Access Control Markup) [21]. Santos et al. provides a complete overview of how the DSA Policy Framework compares to previous approaches. [12].

2.2.1 The PROV Data Model

The DSA Policy Framework needs to consume requests to transmit to give a result. A useful model to represent these requests is the PROV Data Model, which is serialized in RDF via the PROV Ontology [13]. The PROV Data Model is designed to model provenance: a

detailed log of actions including details about who started the action and what the target of the action was. This lends itself well to computable policies because it can effectively model actions that may or not be allowed given the properties of the action.

The PROV Data Model is split among three main classes: An activity, an agent, and an entity. Properties that are specific to an activity are linked to the activity, while properties about the agent are linked to agent. In addition, the three classes are linked to each other via association properties (e.g. *wasAssociatedWith*, *actedOnBehalfOf*, etc.). While the model predicates are past tense, it can be used to effectively model future events as well (e.g. transmitting data in the future). As a result, the PROV Data Model is a good baseline for computable policies.

2.2.2 Computable Policies in the DSA Policy Framework

From an informal analysis of the NTIA Redbook, Santos et al. observed that the language of the policies follow a conditional expression: IF *condition* THEN *PERMIT* or *DENY*, where the *PERMIT* or *DENY* is applied if the condition is satisfied [12]. The conditions of a policy are based on the following attributes:

- Requester: The type of device that is trying to transmit
- Affiliation: The affiliation of the requester (Federal or Non-Federal)
- Frequency: The frequency range that the device will transmit over
- Location: Where the device will transmit
- Time: When the device will transmit

To be evaluated, requests to transmit need to contain these attributes. Figure 2.1 shows an example request containing an activity resource and an agent resource, where an activity resource contains the time a transmission is expected to occur, and an agent resource that lists attributes of the device (location, frequency range, etc.). To support modelling of these attributes in RDF, the request model also makes use of a domain ontology (the DSA Ontology), and external ontologies (PROV-O, SIO [22]). As shown in the diagram, attributes already supported by PROV-O use their respective predicates (*prov:atLocation*),

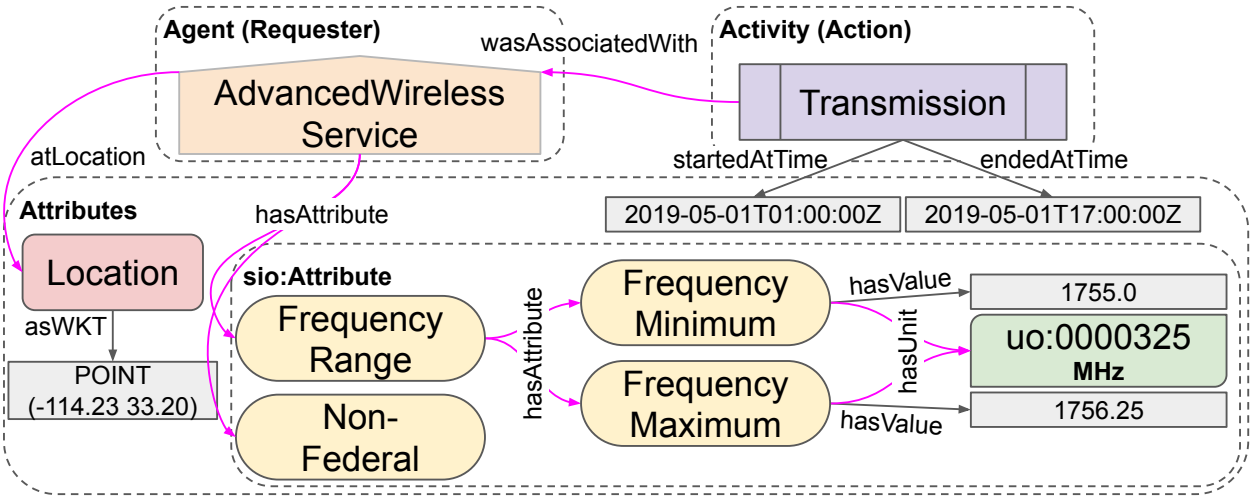

Figure 2.1: A diagram of a sample request used by the DSA Policy Framework. The requester is a Generic Joint Tactical Radio System (JTRS) radio at -114.23, 33.20 (described using Well-Known Text and EPSG:4326) trying to transmit over the 1755-1756.25 MHz range. Reprinted by permission from Springer Nature: H. Santos *et al.*, "A semantic framework for enabling radio spectrum policy management and evaluation," in *The Semantic Web – ISWC 2020*, J. Z. Pan *et al.*, Eds., ser. Lecture Notes in Computer Science, vol. 12507, Cham, Switzerland: Springer Nature, Oct. 2020, pp. 482–498. doi: 10.1007/978-3-030-62466-8_30.

while complex attributes such as frequency range make use of SIO-style modelling on classes the DSA Ontology (*FrequencyRange*, *FrequencyMinimum*, *FrequencyMaxiumum*).

To apply a policy, the DSA Policy Framework classifies requests as instances of a policy to return a *PERMIT* or *DENY* response. OWL is used to enable the domain-specific reasoning through equivalence statements and restrictions to check the attributes of a request. In addition, while policies in the NTIA Redbook follow a conditional expression, many of the complex policies have sub-sections that allow or deny transmissions based on different attributes. This is facilitated in the DSA Policy Framework through extending policies, where a policy may inherit rules and effects from a parent policies in addition to it's own rules.

While the "IF *condition* THEN *PERMIT* or *DENY*" nature of radio spectrum policies lends itself well to automated reasoning, the encoding of data in the reasoning rules rather than direct attributes of the policy is a good example of a graph structure that is awkward for programmers to use. For example, each transmission request is associated with an agent

```
Class: ExamplePolicy
EquivalentTo:
  Transmission
  and (wasAssociatedWith some SpaceToEarthSystem)
  and ((wasAssociatedWith some
          (hasAffiliation some NonFederal))
       or  (wasAssociatedWith some
          (hasAffiliation some Federal)))
  and (wasAssociatedWith some (hasAttribute some
      (FrequencyRange
       and (hasAttribute some
          (FrequencyMaximum
           and (hasValue some xsd:float[<= 403.0f])))
       and (hasAttribute some
          (FrequencyMinimum
           and (hasValue some xsd:float[>= 399.9f]))))))
SubClassOf:
  PermittedActivity, Transmission, Priority_1
```

Figure 2.2: An example policy in Manchester syntax with OWL rules on the PROV model. The policy allows all transmissions in the 399.9 MHz to 403.0 MHz range from a *SpaceToEarth* device class that are affiliated with either Federal or NonFederal activity. The priority level indicates that it overrides other policies.

resource describing the transmission origin, which the model accommodates with restrictions on the *prov:wasAssociatedWith* property. While this pattern is optimized for reasoners, the deep graph structure and overload of properties such as *prov:wasAssociatedWith* means the software needs to check each property and use the shape of it to determine the location of a specific property. As mentioned above, it is difficult for applications to extract information from this sort of structure in a robust manner since they are tightly linked to the ontology.

This motivates the idea of decoupling our application from the ontology using the hybrid modelling approach of Puleston et al.[9] to abstract out representation that is useful for the reasoner. In the end, this gives us a higher-level and domain-specific model of radio spectrum access which we describe in the later sections.

2.3 Domain Specific Languages

A domain-specific language (DSL) is a language aimed towards a specific set of problems, unlike a general-purpose programming language (GPL), such as C++ or Java. The domain in question can vary in scope, with examples such as document styling (CSS) or SQL query generation (LINQ). While there is no formal definition for a DSL, Taha proposed four characteristics of a DSL:

1. The domain is well-defined and central.
2. The notation is clear.
3. The informal meaning is clear.
4. The formal meaning is clear and is implemented.

The first three characteristics make up what is called jargon, the language of domain experts, while the fourth characteristic is what transforms jargon into a DSL [23].

2.3.1 Domain Specific Language Motivation

Mernik et al. identified several common situations that motivate development of a new DSL [24]. While DSLs often require significant implementation effort, we make use of a DSL for the following patterns/reasons:

- Addition of domain-specific notation
- Automation of repeated tasks for developers
- Expression of complex data structures in a straightforward way

The remaining subsections sections briefly explain the reasoning behind these patterns.

2.3.1.1 Addition of Domain-Specific Notation

A major part of application development is the creation of tests cases, which come with example data to verify that an application works. With the RDF serialization formats, the programmer needs to think about the graph model of RDF when creating any example policy. This becomes tedious in the face of many test cases. Something that may lighten

the load on the programmer is a more concise representation of a policy, which allows them to only think about the domain-specific attributes.

This also applies to the end user. The DSA Policy Framework offers several visual tools to view and edit policies, but allowing for description of policies in text form condenses all aspects of the policy into a concise, human-readable form. While a GUI is usually sufficient for editing and viewing domain objects, it becomes tedious to use with repetition, or in bulk. As such, policies can be composed into documents, which is useful for writing/reading multiple policies to/from a file at a time. This also makes the DSL well suited for the document nature of regulations. Although RDF languages such as Turtle allow for description of policies as text, DSLs are closer to the domain, so domain experts (spectrum managers) will not need to learn the concepts of OWL or the syntax of such languages.

2.3.1.2 Automation of Repeated Tasks for Developers

In GPLs, developers often spend time with tasks that are tedious and/or repetitive in nature [24]. In our case, policies in OWL rely on multiple nested restrictions to read attribute types on requests. In addition, policies often contain the same (optional) features: locations, devices, frequency ranges, obligations, and effects. With the domain-specific notation provided by a DSL, the cognitive load of creating many policies is reduced by abstracting out restrictions and URL-specific properties, which are often shared between OWL restrictions. Specifically, a DSL can help developers write the RDF-form of many policies. As mentioned above, this further supports easier testing or demonstration of applications with this automation. Overall, DSLs may enable easier future maintenance of Ontology-Driven Architectures, as they have for general applications [25]. Even for non-developers, this domain-specific nature reduces the difficulty in working with large scale systems [26], where managers will need to work with a large number of policies.

2.3.1.3 Expression of Complex Structures in a Straightforward Way

As mentioned before, directly deriving application data from policies in the knowledge graph would tightly link the ontology to RDF graph traversal code [15]. Since ontologies can undergo many changes during development, all instances of graph traversal code also need to updated often, increasing maintenance complexity. In these cases, having a DSL with a domain-specific model can act has a stable API, containing only domain-relevant content,

while the underlying Knowledge Graph can change to take advantage of the strengths of OWL.

3. DESIGN

With the use of a domain-specific model to represent policies, the model we outline is a hybrid approach, which is mostly motivated by the general structure of computable policies in OWL. Policies make significant use of meta-modeling / punning to enable the complex reasoning that they do, and the policies in the DSA Policy Framework are no exception. Since all instances of policies are OWL classes, a direct approach would instantiate every policy as a Java class, which is infeasible because our underlying knowledge graph is constantly changing. Constantly needing to recompile the code leads to downtime for an application and does not scale. In addition, indirect approaches often lead to the same awkward graph traversal code when developing a domain-specific application. The approaches most relevant to our hybrid approach are HOBO [27] and Mooop [28], but neither framework easily supports meta-modeling. Since a large amount of application data is derived from equivalence statements, the proposed model needs to explicitly manage its hybridization and cannot use either framework.

3.1 Requirements

The main goal of the hybrid modelling is to allow those not familiar with RDF or OWL (such as domain exports) to contribute. The models that they use should describe the original policies in OWL without the representation necessary for reasoning. However, it should still be able to regenerate these components, since these models are the basis for applications to add new data to the underlying knowledge graph. With the design of the policies in the DSA Policy Framework, there are several requirements to consider:

Encapsulate Attribute Restrictions Transmission requests are associated to a policy by the reasoner checking whether the attributes of a request are in the equivalent class of the policy. As such, the domain models need to store the restrictions on attributes mentioned above: the device class, affiliation, frequency range, location of origin, and the desired time of transmission.

Encapsulate Policy Effects When a decision is made, the request inherits the effects associated with the policy: Whether the activity is permitted or denied, and any additional

constraints on transmission (denoted "Obligations"). The domain models need to store the relevant effect and obligation for each policy.

Encapsulate Policy Extension An important part of the policy is the ability to describe extensions to it, to adequately represent the sub-sections of NTIA Redbook policies. The models need to be able to store which parent policy a policy inherits from.

Support Translation to Other Representations The motivations for this goal are mostly outlined in section 2.3.1, with the DSL. However, beyond the DSL, the ability to translate to other representations may facilitate faster exploration of data, especially with formats that focus on different aspects of the policy (e.g. the logic with the Common Logic Interchange Format).

Contain Domain-Specific Information OWL is not a programming language [14, 9]. As a result, directly integrating it into a domain-specific application is awkward and demands significant maintenance effort on behalf of the developers. Since the domain models provide an interface to the knowledge graph, it should be simple for developers to interpret the model and extract necessary data without thinking about RDF or OWL.

3.2 Radio Spectrum Policies in Software

There are many direct and indirect modelling approaches that integrate with Java. In addition, the hybrid modelling of Puleston et al. also used Java-like classes to model their domain [9]. In this work, we used Scala, a JVM language with many similarities to Java. While Scala does have some differences from Java, the hybrid models we show below can be easily converted into Java classes (e.g. *null* values or Java 8's Optional for optional values).

To model the domain, each policy is an instance of a *Policy* class, as seen in Figure 3.1. Each policy has an identifier, a human-readable name (the label) and a description of a policy in English. While these have direct counterparts in the ontology, the restrictions that describe the scope of a policy abstract out the OWL and are direct properties of the class. This provides a cleaner API than reading the overloaded *wasAssociatedWith* properties, and is generally more natural to the programmer than the OWL. Each restriction type is also modeled as a class. For example, there is a *FrequencyRange* case class that stores

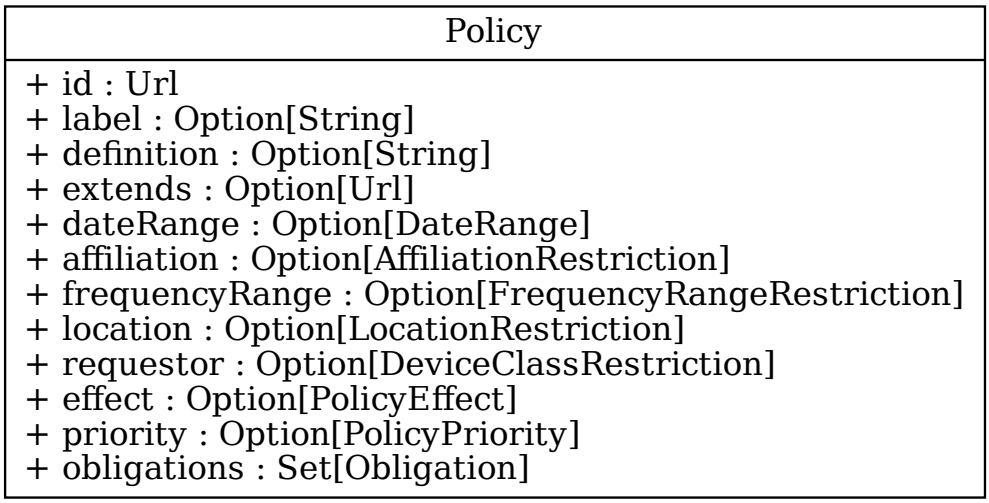

Figure 3.1: A UML diagram of a portion of the *Policy* class, with the attributes to describe what requests it applies to.

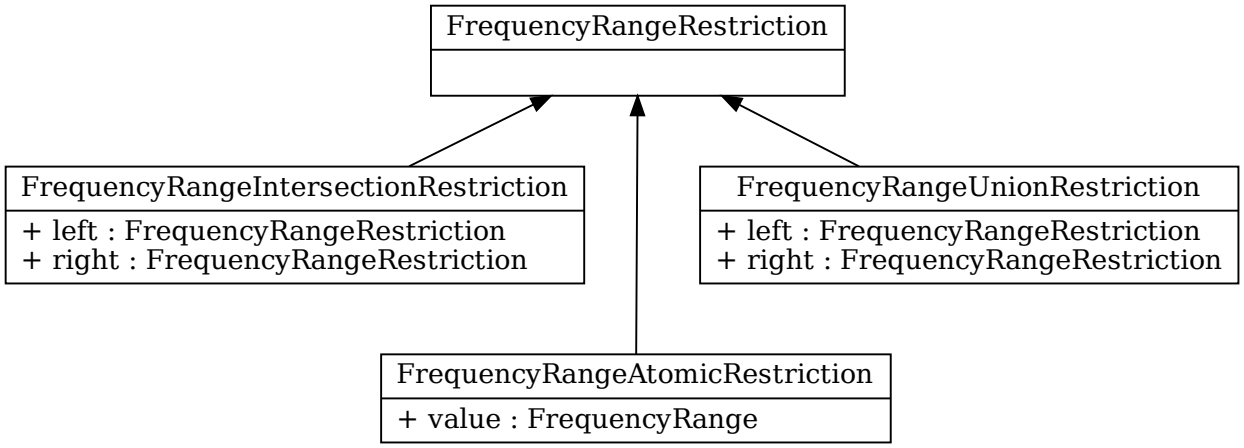

Figure 3.2: An example subclass hierarchy of restrictions on frequency ranges. Each "agent" attribute (*Affiliation*, *FrequencyRange*, *Location*, *DeviceClass*) has their own subclass hierarchy for group unions and intersections.

the minimum frequency, maximum frequency, and the unit of the range (which defaults to Megahertz). In addition to the restrictions, the policy has the effect of the policy as direct properties. Both the *PolicyEffect* and *PolicyPriority* are effectively enumerations (implemented as sealed traits in Scala), so that the programmer knows what values the properties are limited to. Finally, the obligations are represented as a set of strings (boxed in an Obligation object to support further expansion to Obligation capabilities).

An additional component that needed to be modeled was combinations of restrictions.

While a policy requires that a request satisfy all restrictions to be applicable, a restriction may be made up of multiple parts with the *unionOf* OWL property, which one of the two restrictions to be fulfilled to fulfill the main restriction (an example of this is shown in Figure 2.2). Likewise, an *intersectionOf* counterpart exists, where both restrictions need to be fulfilled. In the *Policy* case class, this is modeled through recursive data types, as seen in Figure 3.2. Both union and intersection variants have left and right fields that take the base restriction as values. The base case for the recursive data type is a *AtomicRestriction* subclass that contains a field to contain a single restriction.

Since most policies are broken up into smaller policies in the DSA Framework, being able to extend policies is also an important feature. Instead of storing a parent policy directly, the model chooses to delegate this to the knowledge graph. The benefit of this delegation is that it allows the model to focus on how a child policy extends a parent policy. While the model does not directly store the restrictions of a parent policy, this is something that can be extracted into a separate *Policy* object when needed by the programmer. In addition to policy extensions, policy attributes (or the specific value restrictions) are delegated to the knowledge graph. Entities such as affiliations or radio device types often contain extra modelling on behalf of an ontologist, and this delegation allows for flexibility in how the ontologist chooses to model them. On the other hand, this limits what the domain-specific application can explore (discussed in more detail in chapter 6).

3.3 Domain-Specific Language Design

In the DSL, each policy is represented as a set of clauses (similar to SQL), with an example in Figure 3.3. A policy starts with the keyword *POLICY* followed by the label, identifier, and description in quotes. This is then followed by the clauses representing the restrictions, where each keyword (*EXTENDS*, *DEVICECLASS*, *AFFILIATION*, etc.) add a specific restriction type to the policy. The DSL also supports unions and intersections with the *and* and *or* keywords between restrictions of the same type. Finally, the policy definition ends with the clases defining the effects. The *PERMIT* or *DENY* keyword is used to define the immediate effect, while *OBLIGATION* is used to add string obligations to the policy. Likewise, the *PRIORITY* keyword is used to add a numeric priority the policy.

To improve the readability of the policy, the DSL includes some optional keywords

```
1 POLICY [Example Policy](http://example.org/ExamplePolicy)
2 "This is an example policy."
3 IF TR
4    EXTENDS(dsa-policy:NTIARedBookUSTransmission)
5    BY DEVICECLASS(dsa-t:SpaceToEarthSystem)
6    FOR (AFFILIATION(dsa-t:NonFederal)
7         or AFFILIATION(dsa-t:Federal))
8    ON FREQRANGE(399.9, 403.0, Mhz)
9 THEN PERMIT
10   PRIORITY 1
```

Figure 3.3: The example policy of Figure 2.2 written in the DSL, where indentation is used to enhance readability. The shortened URIs represent objects delegated to the knowledge graph in the hybrid modelling. Pink signifies important keywords, while blue signifies optional keywords. Orange highlights unions and intersections of restrictions.

to add more structure. The *IF TR* and *THEN* denote the blocks of restriction and effect clauses, while attribute keywords (BY, FOR, ON, etc.) hint at what an attribute means to a radio transmission request. A more formal description of the DSL is available in Appendix 6.2.

4. IMPLEMENTATION

To apply hybrid modelling, we implemented a Scala library (we which call *DsaModels*) that translates from the RDF to domain models and from the domain models to RDF. Apache Jena [17] is used to navigate the graph and generate different parts of the policy. A benefit of the single library is that the graph is traversed through at most once. In addition, there is one location where the code is linked to the graph. This effectively decouples other application logic from the knowledge graph, as any changes the the graph structure only results in changes to the library, while the *Policy* case class remains stable.

This library was also used to enable translation from the domain model to the DSL and back. Parsing of the DSL was handled via the Scala Standard Parser Combinator Library [29] (specifically, the recursive-descent parser combinator for regex parsing). Since the translations relies on the domain models rather than the knowledge graph, it is relatively simple to add many different serialization formats for a policy.

Beyond providing a translation interface, the library also provides utilities to manipulate attribute restrictions on policies. When dealing with multiple restrictions in a *unionOf* or a *intersectionOf*, RDF represents them in a function list format (with first and rest). However, depending on the order of insertion, the resulting list may not give a flat structure well suited for traversal. To deal with this, the library provides a *flatten* function that moves elements to the rest property when it can (while maintaining equivalent logic). Furthermore, with the Cats library for Scala (which supports more mathematical paradigms for Scala), Policies can be combined (combining their restrictions in an intersection) using a Semigroup type. This is useful for generating a child policy with the all of the restrictions of a parent policy.

5. EVALUATION

To evaluate our work, we first compare to how the implementation of the hybrid approach fulfills the design requirements outline in Chapter 3. In addition, we do a separate evaluation of the DSL by comparing how concise it is compared the RDF by comparing how many lines it takes to represent a policy in either serialization. We evaluate each goal separately:

5.1 Encapsulate Attribute Restrictions

The domain models should encapsulate the features that the DSA Ontology is able to. We base our comparison of supported features on the Santos et al., which evaluates the DSA Framework in a similar manner, as seen in Table 5.1 [12].

Requester-specific attributes are direct attributes of origin of a transmission request. This involves the device type that is trying to transmit and who the device is transmitting for. Since the device type and the organization behind a transmission are represented as subjects in the ontology, both the DSA Framework and the Domain models use the direct URLs to refer to them, and as such can represent such restrictions. However, the task of dealing with multiple related requesters (categorized as "Dependency") is much more difficult, as it involves requires looking at multiple requests at once. This feature is still incomplete in the DSA Framework and thus, the domain models are also unable to model multiple related requestors. Finally, a licensee (associating a request with a specific licence) is done through having a resource in the knowledge graph for the license. As such, the support for licenses is partial, and the domain models can also use the URL to match the DSA Framework in this aspect.

Affiliation attributes are slightly different from the attributes such as the organization, as they represent the general organization that are associated with the device (e.g. military systems and federal systems). Santos et al. lists support for these as "partial", as they currently require named entities in the knowledge graph (having a specific URL for Federal, Non-Federal and Military). The domain model models an affiliation with a URL as well, so support for affiliations is on par with the DSA Framework (full named requester support and partial generic requester support).

Policy frequency ranges are supported in both the DSA Framework and the hybrid

Table 5.1: A table comparing the ability of the DSA Framework, Domain Models and the DSL to represent restrictions on a transmission request. "Delegated" means that the restriction on that feature is left in the knowledge graph and a reference is maintained via a URL. Ideally, the columns should match up as much as possible, meaning that hybrid models can model computable policies.

	Implemented		
	DSA Framework	**Hybrid Models**	**DSL**
Requester-Specific			
Device	yes	yes	yes
Organization	yes	yes	yes
Dependency	no	no	no
Licensee	partial	partial	partial
Affiliation			
Generic Requesters	partial	partial	partial
Named Requesters	yes	yes	yes
Frequencies			
Frequency Range	yes	yes	yes
Single Frequency	yes	yes	yes
Named Bands	yes	yes	yes
Units	yes	yes	yes
Time			
Timezones	yes	yes	yes
Policy Validity	yes	yes	yes
Locations			
Named Locations	yes	yes	yes
Relative Locations	no	no	no
Polygons / Circles	yes	delegated	delegated
Geographical Rules			
Specific Location	yes	delegated	delegated
List of Locations	yes	delegated	delegated

modelling approach. The DSA Framework uses the XSD (to set maximum and minimum) and SIO (to set units) ontologies to impose restrictions on the range, while the hybrid modelling approach contains this information in a simple *FrequencyRange* object. One feature of frequency ranges to note are "named bands", which represent a constant name associated with a range (e.g. the AWS-3 band). While it is possible to represent this by name, a request to transmit is not based on the named band, but rather a specific frequency range it transmits on. As such, both the DSA Framework and the hybrid models support named bands as normal frequency ranges.

The time range that a policy applies to is simple to represent, as it only requires a start and end time ("Policy Validity"). As such, both the framework and hybrid model support this feature. The other major component is the timezone frame of reference for the start and end times. However, this is mostly automatically handled. For example, the both the DSA Framework and DSL uses timestamps based on ISO 8601 (a standard for string timestamps), while the hybrid model relies on Java's standard time library to represent restrictions.

While the hybrid model and DSL share the feature parity of the DSA Framework for the attributes above, one area where they are lacking is the representation of locations. While the hybrid model does support representing specific polygons and lists of location, by default *DsaModels* chooses to delegate this representation to the knowledge graph when converting from RDF to objects. This means that the hybrid model is aware of the reference, but leaves up extracting this information to another query from the knowledge graph. As such, some specific features of locations is listed as "delegated". It should be noted that this is characteristic of hybrid models. Hybrid models aim to improve the development experience by pushing the ontological representation to the corners of the model, which allows it to keep the strengths of the original RDF model in place. One such example of this is the affiliation of a requester. A UI to edit policies only really needs to know which affiliations are which, which can be done by a simple comparison of a URI. However, the affiliation may contain more properties that are interesting to the user, such as a description of what the affiliation represents. Ontologies usually include this detail to add provenance to the data: one of the goals of the semantic web. By keeping a reference to the affiliation, an application may be able to take advantage of the extra modelling in the graph, which is a strength of the underlying RDF model.

Since the hybrid models are the DSL are able to keep up with the DSA Framework in their representation (all columns are mostly the same), hybrid modelling is feasible to modelling the rules on which the policies are applied. As such, *DsaModels* successfully encapsulates attributes restrictions.

5.2 Encapsulate Policy Effects

Similar to Chapter 5.1, we compare if the hybrid modelling approach can represent the effects of a policy compared to the DSA Framework, as seen in Table 5.2.

Table 5.2: A table comparing the ability of the DSA Framework, the hybrid models, and the DSL to represent the effect, precendence and obligations of a policy, for quick reference. Since *DsaModels* is able to model the same attributes the DSA Framework can, it is able to capture the essence of the complex domain.

	Implemented		
	DSA Framework	**Hybrid Models**	**DSL**
Effect	yes	yes	yes
Precedence	yes	yes	yes
Obligations	partial	partial	partial

The effect of a policy is the ultimate decision that decides whether someone is allowed to transmit or not. In the DSA Framework, this is simply represented as two resources representing "permit" and "deny" respectively. These have equivalents in the hybrid models, but are essentially represented as Scala enumerations. The benefit of this is that it gives the developer type-safety: reassurance that the value of the property is one of the two and not something else. Precedence is implemented as an integer ranging from 1 to 6 for both the DSA Framework and hybrid model. Since the DSL has distinct clauses for each effect property, both have full support for representing policy effects and precedence.

Support for obligations is partial. In writing (i.e. the NTIA policy manuals), these are equivalent to behaviors that the requester must obey during transmission. Since the DSA Framework quotes these in strings for obligation representation, the hybrid models and DSL follows suit by using strings for representation, giving it the partial ability to represent them (no support for directly reasoning on them).

Since the hybrid model and DSL can represent the policy effects as well as the original RDF models in the DSA Framework, the hybrid approach can successfully encapsulate policy effects.

5.3 Encapsulate Policy Extension

The DSA Framework breaks down complex policies by extending parent policies with child policies that add restrictions for a policy to be applied. This is done through using both the equivalent class and sub class OWL properties. In the hybrid model, this is simple done through a "extends" property in the policy object, which once again delegates the

```
1 (forall (fmin Freq) (fmax Freq)
2         (req Requester)
3         (aff Affiliation)
4         (eff Effect)
5         (pri Priority)
6   (if (and
7         (>= fmin 403.0) (<= fmax 399.9)
8         (= req dsa-t:SpaceToEarthSystem)
9         ((= aff dsa-t:Federal) or (= aff dsa-t:NonFederal)))
10       (and
11        (= eff Permit)
12        (= pri 1))))
```

Figure 5.1: The example policy of Figure 2.2 written in CLIF. Note that this serialization focuses on the logic, and is missing information about the policy (e.g. the name of the policy).

relationship between parent and child policies to the knowledge graph. However, given multiple policy objects, *DsaModels* provides functionality that makes it to generate a child policy combined with all parent policies. With this in mind, the hybrid modelling approach successfully encapsulates policy extensions.

5.4 Support Translation to Other Representations

DsaModels supports traversing an RDF graph to generate the respective policy objects. The DSL was designed to follow the domain-specific nature of the hybrid models, so translations from the hybrid models to the DSL (and back) just involves reading from the fields of the policy object instead of needing to re-traverse the RDF. While this gives the necessary evidence that the library support this goal, this is further supported by work (within the DSA Framework, but not part of this thesis) to support one-way translation from the hybrid models to the Common Logic Interchange Format (CLIF), which an example of is given in Figure 5.1.

CLIF is a first-order logic language that uses Lisp-like syntax to specify conditions and effects. As a result, it excels at describing the functional aspects of the policy, but also loses much of the important information about the policy. However, this limited scope is what makes it useful for domain experts (radio spectrum policy managers) to understand what

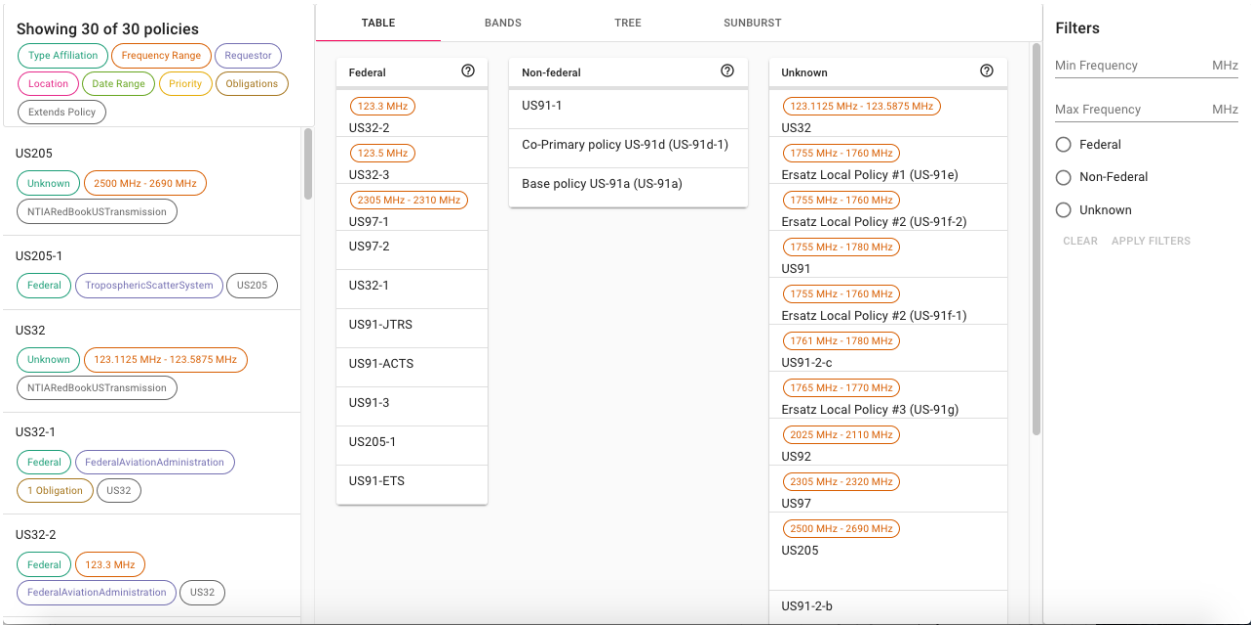

Figure 5.2: A screenshot of a web application to view radio spectrum policies. The application uses the DSA Models library to translate RDF into the hybrid models, which are then sent to the page via a GraphQL API.

the policy actually does. This is also a goal of the DSL: by using short clauses about the properties, it should be much easier to view and edit policies in bulk. Rather than having the user repeatedly click through pages for each policy and using buttons to change aspects of the policy (which is tedious for the user), the user is able to search through a text document and edit the policy at will.

The translation interface for CLIF is also based on the hybrid models, as it is easy to read from the properties of the policy object. Since both serializations are able to take advantage of the original code translating from the RDF into the hybrid models, this goal is achieved in both theory and practice.

5.5 Contain Domain-Specific Information

Since the DSA Framework aims to use class equivalence conditions to associate policies with a radio transmission request, the attributes of a policy are deeply nested in OWL semantics. The hybrid models are designed to abstract out much of the representation that OWL provides, to augment the knowledge graph with objects that applications can easily

use. As shown in Chapter 3, instead of equivalence conditions and OWL subclassing, the attributes of a policies are directly stored as attributes of a policy object. To further support the domain-specific nature of the hybrid models, the DSA Models library has been used to develop applications (within DSA Framework, but not part of the thesis) to view and edit policies. As such, the hybrid models successfully contain domain-specific information.

5.6 DSL Evaluation

To evaluate how concise the DSL representation of a policy is, we compares how many lines it takes to represent some NTIA Redbook policies in the DSL compared to Turtle and JSON-LD. In this case, the Turtle and JSON-LD serialization are semantically equivalent, as they are both based on the same OWL rules. The DSL is not semantically equivalent in that sense, but there is a one-to-one mapping from the DSL to the OWL rules. To keep this a fair comparison, all serializations will be limited to what the DSL can represent and delegate to the knowledge graph: restrictions and effects of parent or child policies (separately). For example, locations are limited to their URL reference, as the DSL delegates them to the knowledge graph. The line count for the Turtle serialization will be based on the default output Turtle output of Apache Jena without prefix definitions. Likewise, we will use the compact form of JSON-LD. Since Jena outputs a *@context* node for the compact JSON-LD, we removed this node in the output while keeping the shortened representation. Finally, the serialization for the DSL includes all optional keywords.

As seen in Figure 5.3, the DSL representation always ends up being the shortest representation of a policy. On average, it takes 61.194 lines to represent a policy in Turtle and 60.179 lines in JSON-LD, whereas by comparison it takes 7.208 lines in the DSL. In extreme cases, both the Turtle and JSON-LD representations even end up with more than 100 lines for a policy. While there are quite a few factors playing a role in these numbers, the most notable one is the how much boilerplate is needed for a single attribute. Attributes in the OWL representation need a lot of OWL-specific semantics that enable a reasoner to associate requests with a policy. As a result, each new attribute can add upwards of 5+ new triples (subject, predicate, object) that contribute to the line count, which can vary depending on the type of attribute. Since the general structure of a frequency range (see Figure 2.1) is so nested, this ends up contributing the most to the line count.

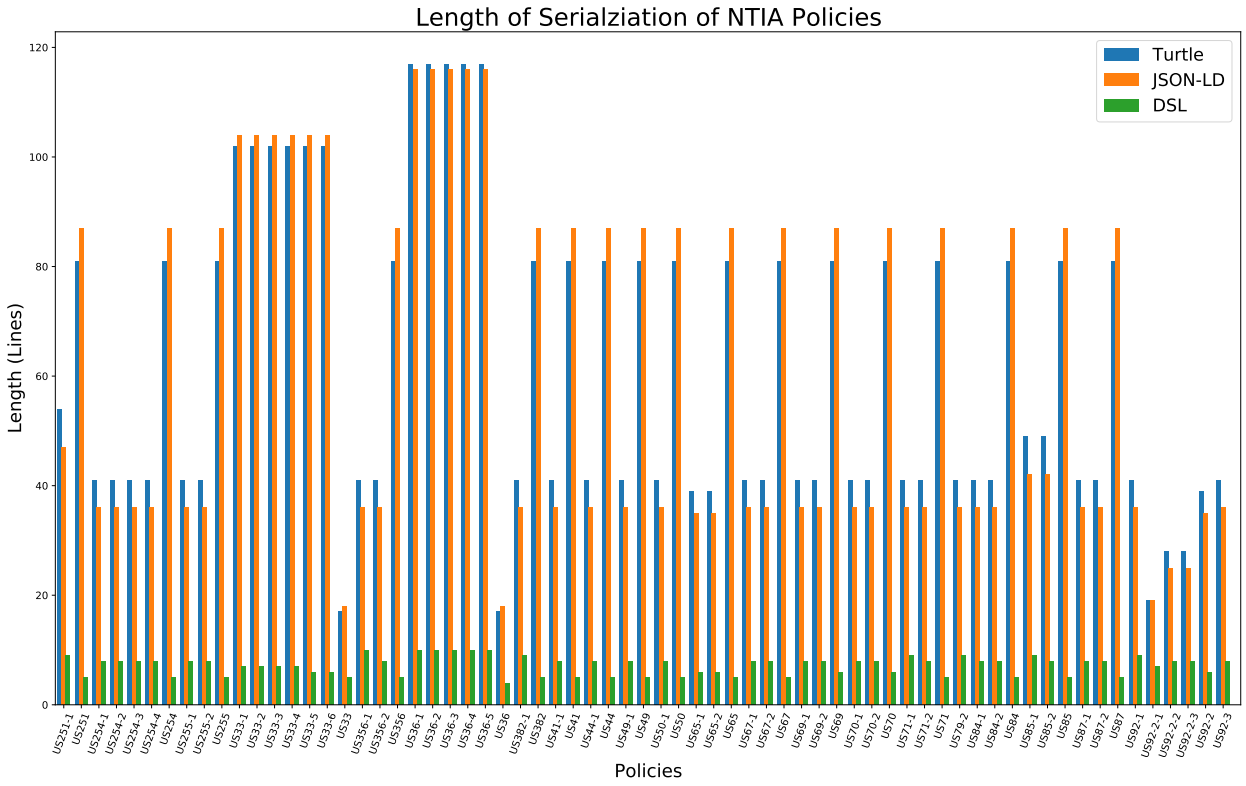

Figure 5.3: A grouped bar graph comparing the length of some policies from the NTIA Redbook in Turtle versus the DSL. The average is 61.194 **lines for Turtle,** 60.179 **lines for JSON-LD, and** 7.208 **lines for the DSL.**

As seen in Figure 5.3, the DSL efficient in how it represents each property. As such, the DSL is successfully able to represent a policy in a concise manner.

6. CONCLUSION

This thesis applies a hybrid approach to modelling computable policies in domain-specific software. Furthermore, it introduces the concept of using a domain-specific language (DSL) to describe a domain originally modelled in RDF, as it could work as a more useful serialization for an end-user or developer.

The work was evaluated by evaluating for feasibility by how accurately the hybrid modelling could encapsulate the semantics of the original RDF model in the radio spectrum management domain. With some delegation to the knowledge graph, it was able to represent the same features, while providing a better application development experience, as the hybrid models are more easily maintained and requires less awkward code to extract data from [9]. Additionally, the DSL was evaluated by comparing how many lines it needed to represent a policy compared to Turtle, a common RDF serialization format. As the DSL was designed to be readable to non-ontology experts, this makes it a useful serialization, potentially making it easier to view or edit policies in bulk.

6.1 Limitations

It is important to acknowledge that a hybrid approach still requires a significant amount of work on behalf of the developer. The programmer needs to ensure the core classes maintain a one-to-one mapping with the ontology, which can be done through thorough testing. Likewise, DSL development is generally regarded as a difficult task requiring large amounts of work [24]. However, this form of mapping is not entirely new in general software development, as similar software such as Object-Relational Mappings (ORMs) is commonly used with relational databases. Unfortunately, knowledge graphs are still missing a common framework for object mapping. While there are approaches to handle mapping RDF triples to objects (e.g. Trinity RDF [30]), they are still limited in that they operate on direct predicates and the result object is still tightly linked to the knowledge graph.

It is also possible to argue that the direct components of a hybrid model constrains expressivity. While this is true, developing domain-specific applications already puts constraints on a model, since the software needs to parse the model. In that sense, the hybrid approach also reduces constraints on expressivity because there is one point of change when

the model is changed.

One final limitation is on the ability to access the indirect model. While the hybrid models are able to point a user to where they can do further exploration, the core classes hide direct access entities in the knowledge graph. As such, the programmer usually must make some interface available to the user if they want to explore more of the domain. This usually comes in the form of choosing to directly model these entities, a separate query interface, or using separate domain-neutral exploration software, as Puleston et al. has proposed [9].

6.2 Future Work

In terms of domain-specific future work, some aspects of *DsaModels* can be further improved upon (e.g. fully support translating location to and from RDF in the hybrid models). More generally, some potential future work includes applying this approach to other different domains, as hybrid modelling has been applied to relatively few domains. While the work with the DSA Framework was able to show the ability of the hybrid modelling to encapsulate computable policies, this could potentially be extended to the PROV data model. A generic hybrid modelling approach for the PROV data model would go a long way in speeding up development of applications with complex domains and further reduce the workload on the developer.

REFERENCES

[1] *RDF 1.1 Concepts and Abstract Syntax*, W3C, Feb. 2014. Accessed: Apr. 1, 2021. [Online]. Available: http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/

[2] E. F. Kendall and D. L. McGuinness, *Ontology Engineering*. San Rafael, CA, USA: Morgan & Claypool, 2019. DOI: 10.2200/S00834ED1V01Y201802WBE018.

[3] *OWL 2 Web Ontology Language*, W3C, Dec. 2012. Accessed: Apr. 1, 2021. [Online]. Available: http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/

[4] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang, "HermiT: An OWL 2 reasoner," *J. Automat. Reason.*, vol. 53, no. 3, pp. 245–269, Apr. 2014. DOI: 10.1007/s10817-014-9305-1.

[5] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical OWL-DL reasoner," *J. of Web Semantics*, vol. 5, no. 2, pp. 51–53, Jun. 2007. DOI: 10.1016/j.websem.2007.03.004.

[6] *RDF 1.1 XML Syntax*, W3C, Feb. 2014. Accessed: Apr. 1, 2021. [Online]. Available: http://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/

[7] *JSON-LD 1.1*, W3C, Jul. 2020. Accessed: Apr. 1, 2021. [Online]. Available: https://www.w3.org/TR/2020/REC-json-ld11-20200716/

[8] *RDF 1.1 Turtle*, W3C, Feb. 2014. Accessed: Apr. 1, 2021. [Online]. Available: https://www.w3.org/TR/2014/REC-turtle-20140225/

[9] C. Puleston, B. Parsia, J. Cunningham, and A. Rector, "Integrating object-oriented and ontological representations: A case study in Java and OWL," in *The Semantic Web - ISWC 2008*, A. Sheth *et al.*, Eds., ser. Lecture Notes in Computer Science, vol. 5318, Berlin, Germany: Springer-Verlag, Oct. 2008, pp. 130–145. DOI: 10.1007/978-3-540-88564-1_9.

[10] *SPARQL 1.1 Overview*, W3C, Mar. 2013. Accessed: Apr. 1, 2021. [Online]. Available: https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/

[11] D. R. Kuhn, E. J. Coyne, and T. R. Weil, "Adding attributes to role-based access control," *IEEE Computer*, vol. 43, no. 6, pp. 79–81, Jun. 2010. DOI: 10.1109/MC.2010.155.

[12] H. Santos *et al.*, "A semantic framework for enabling radio spectrum policy management and evaluation," in *The Semantic Web – ISWC 2020*, J. Z. Pan *et al.*, Eds., ser. Lecture Notes in Computer Science, vol. 12507, Cham, Switzerland: Springer Nature, Oct. 2020, pp. 482–498. DOI: 10.1007/978-3-030-62466-8_30.

[13] *PROV-O: The PROV Ontology*, W3C, Apr. 2013. Accessed: Apr. 1, 2021. [Online]. Available: http://www.w3.org/TR/2013/REC-prov-o-20130430/

[14] S. Baset and K. Stoffel, "Object-oriented modeling with ontologies around: A survey of existing approaches," *Int. J. of Softw. Eng. and Knowl. Eng.*, vol. 28, no. 11n12, pp. 1775–1794, Nov. 2018. DOI: 10.1142/S0218194018400284.

[15] A. Rector, M. Horridge, L. Iannone, and N. Drummond, "Use cases for building OWL ontologies as modules: Localizing, ontology and programming interfaces & extensions," in *Proc. of 4th Int. Workshop on Semantic Web Enabled Softw. Eng. (SWESE-08)*, Karlsruhe, Germany. Oct. 2008. Accessed: Apr. 1, 2021. [Online]. Available: https://www.researchgate.net/publication/228964712_Use_Cases_for_Building_OWL_Ontologies_as_Modules_Localizing_Ontology_and_Programming_Interfaces_Extensions

[16] A. Kalyanpur, D. Pastor, S. Battle, J. Padget, and G. Maurer, "Automatic mapping of OWL ontologies into Java," in *Proc. of 16th Int. Conf. on Softw. Eng. and Knowl. Eng. (SEKE)*, Jun. 2004, pp. 98–103.

[17] Apache Jena. (2021). The Apache Software Foundation. Accessed: Apr. 1, 2021. [Online]. Available: https://jena.apache.org/index.html

[18] J.-B. Lamy, "Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies," *Artif. Intell. in Med.*, vol. 80, pp. 11–28, Jul. 2017. DOI: 10.1016/j.artmed.2017.07.002.

[19] D. Martin *et al.*, "Bringing semantics to web services: The OWL-S approach," in *Semantic Web Services and Web Process Composition*, J. Cardoso and A. Sheth, Eds., ser. Lecture Notes in Computer Science, vol. 3387, Berlin, Germany: Springer-Verlag, Jul. 2005, pp. 26–42. DOI: 10.1007/978-3-540-30581-1_4.

[20] *Semantic Web Services Langauges (SWSL)*, W3C, Sep. 2005. Accessed: Apr. 1, 2021. [Online]. Available: http://www.w3.org/Submission/2005/SUBM-SWSF-SWSL-20050909/

[21] *eXtensible Access Control Markup Language (XACML) Version 3.0*, OASIS, Jan. 2013. Accessed: Apr. 1, 2021. [Online]. Available: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

[22] M. Dumontier *et al.*, "The semantic science integrated ontology (SIO) for biomedical research and knowledge discovery," *J. of Biomed. Semantics*, no. 5, Mar. 2014, Art. no. 14. DOI: 10.1186/2041-1480-5-14.

[23] W. Taha, "Plenary talk iii domain-specific languages," in *2008 Int. Conf. on Computer Eng. Syst.*, Nov. 2008, pp. xxiii–xxviii. DOI: 10.1109/ICCES.2008.4772953.

[24] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, Dec. 2005. DOI: 10.1145/1118890.1118892.

[25] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: A roadmap," in *Proc. of the Conf. on The Future of Softw. Eng.*, Limerick, Ireland: Association for Computing Machinery, May 2000, pp. 73–87. DOI: 10.1145/336512.336534.

[26] J. Gray, K. Fisher, C. Consel, G. Karsai, M. Mernik, and J.-P. Tolvanen, "Dsls: The good, the bad, and the ugly," in *Companion to the 23rd ACM SIGPLAN Conf. on Object-Oriented Program. Syst. Languages and Appl.*, Nashville, TN, USA: Association for Computing Machinery, Oct. 2008, pp. 791–794. DOI: 10.1145/1449814.1449863. 30

[27] C. Puleston and B. Parsia, "The HOBO hybrid modelling framework," in *OWL: Experiences and Directions (OWLED)*, P. Kilnov and M. Horridge, Eds., May 2012. Accessed: Apr. 1, 2021. [Online]. Available: http://ceur-ws.org/Vol-849/paper_15.pdf

[28] C. Frenzel, B. Parsia, U. Sattler, and B. Bauer, "Mooop – a hybrid integration of OWL and Java," in *Adv. Inf. Syst. Eng. Workshops*, C. Salinesi and O. Pastor, Eds., ser. Lecture Notes in Business Information Processing, vol. 83, Berlin, Germany: Springer-Verlag, Jan. 2011, pp. 437–447. DOI: 10.1007/978-3-642-22056-2_47.

[29] *Scala-parser-combinators*. (2019). Scala. Accessed: Apr. 1, 2021. [Online]. Available: https://github.com/scala/scala-parser-combinators

[30] *Trinity RDF*. (2019). Semiodesk GmbH. Accessed: Apr. 1, 2021. [Online]. Available: https://trinity-rdf.net/

APPENDIX A DOMAIN-SPECIFIC LANGUAGE EBNF

This appendix contains EBNF rules that describe the DSL (which is LL(5) due to the frequency ranges). We define a few basic rules:

- `<timestamp>` is a timestamp string as defined by ISO 8601.
- `<literal>` is a sequence of characters, including whitespace characters.
- `<url>` is a url string as defined by WHATWG.
- `<number>` includes string representations of floats or integers

The rules for the DSL are as follows (case-insensitive):

⟨*policy*⟩ ::= ⟨*policy-clause*⟩ ⟨*conditions*⟩ ⟨*results*⟩

⟨*policy-clause*⟩ ::= '`POLICY`' [⟨*policy-name*⟩] ⟨*policy-id*⟩ [⟨*definition*⟩]

⟨*policy-name*⟩ ::= '`[`' ⟨*literal*⟩ '`]`'

⟨*policy-id*⟩ ::= '`(`' ⟨*url*⟩ '`)`'

⟨*definition*⟩ ::= '`"`' ⟨*literal*⟩ '`"`'

⟨*conditions*⟩ ::= ['`IF TR`'] ⟨*condition-clause*⟩*

⟨*condition-clause*⟩ ::= ⟨*extend-clause*⟩
| ⟨*affiliation-clause*⟩
| ⟨*frequency-clause*⟩
| ⟨*location-clause*⟩
| ⟨*device-class-clause*⟩
| ⟨*from-clause*⟩
| ⟨*until-clause*⟩

⟨*extend-clause*⟩ ::= '`EXTENDS`' ⟨*url*⟩

⟨*affiliation-clause*⟩ ::= ['`FOR`'] ⟨*url*⟩

⟨*frequency-clause*⟩ ::= ['`ON`'] ⟨*frequency-restriction*⟩

⟨*location-clause*⟩ ::= ['`AT`'] ⟨*location-restriction*⟩

⟨*device-class-clause*⟩ ::= [‘BY’] ⟨*device-class-restriction*⟩

⟨*from-clause*⟩ ::= ‘FROM TIMESTAMP(’ ⟨*timestamp*⟩ ‘)’

⟨*until-clause*⟩ ::= ‘UNTIL TIMESTAMP(’ ⟨*timestamp*⟩ ‘)’

⟨*location-restriction*⟩ ::= ‘LOCATION(’ ⟨*url*⟩ ‘)’

⟨*device-class-restriction*⟩ ::= ‘DEVICECLASS(’ ⟨*url*⟩ ‘)’

⟨*frequency-restriction*⟩ ::= ‘FREQRANGE(’ ⟨*number*⟩ ‘,’ ⟨*number*⟩ [‘,’ ⟨*unit*⟩] ‘)’
| ‘FREQVALUE(’ ⟨*number*⟩ [‘,’ ⟨*unit*⟩] ‘)’

⟨*results*⟩ ::= [‘THEN’] ⟨*result-clause*⟩*

⟨*result-clause*⟩ ::= ⟨*effect-clause*⟩
| ⟨*priority-clause*⟩
| ⟨*obligation-clause*⟩

⟨*effect-clause*⟩ ::= ‘PERMIT’ | ‘DENY’

⟨*priority-clause*⟩ ::= ‘PRIORITY’ ⟨*number*⟩

⟨*obligation-clase*⟩ ::= [‘WITH’] ‘OBLIGATION "’ ⟨*literal*⟩ ‘"’

⟨*unit*⟩ ::= ‘GHz’
| ‘GigaHertz’
| ‘GigaHZ’
| ‘MHz’
| ‘MegaHertz’
| ‘MegaHZ’
| ‘kHZ’
| ‘KiloHertz’
| ‘HZ’
| ‘Hertz’

APPENDIX B PERMISSIONS

B.1 Permissions for Section 2.2

This file contains license details and terms and conditions for the reproduction of material used in Section 2.2 of this thesis.

File name: `RightsLink Printable License.pdf`

File type: Portable Document Format (PDF)

File size: 52.3 KB

Required application software: Any standard PDF viewer

Special hardware requirements: None