5-20-2011

A Geospatial Service Model and Catalog for Discovery and Orchestration

Elias Ioup
University of New Orleans

A Geospatial Service Model and Catalog for Discovery and Orchestration

A Dissertation

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Engineering and Applied Science
Computer Science

by
Elias Zayd Khalil Ioup

B.S. University of Chicago 2003
M.S. University of New Orleans 2006

May, 2011

Contents

Acknowledgements

I would like to acknowledge and thank the following people for all the help they have provided me in completing my doctoral research. First, thanks are due to my advisor, Dr. Mahdi Abdelguerfi, and the members of my committee; Dr. Shengru Tu, Dr. Golden Richard, Dr. John Sample, and Dr. Huimin Chen; for their research guidance and support. I would like to thank Dr. George Ioup and Dr. Juliette Ioup for their advice, encouragement, and editorial assistance in writing this work as well as instigating my entire PhD process. I would also like to thank my colleagues at the Naval Research Laboratory at Stennis Space Center for initiating my interest in geospatial computing and their support as I worked on my doctoral studies. Most of all, I would like to thank my wife Sarah and my mother Georgette for all their help, support, and tolerance through this long process.

List of Figures

List of Tables

Abstract

The goal of this research is to provide a supporting Web services architecture, consisting of a service model and catalog, to allow discovery and automatic orchestration of geospatial Web services. First, a methodology for supporting geospatial Web services with existing orchestration tools is presented. Geospatial services are automatically translated into SOAP/WSDL services by a portable service wrapper. Their data layers are exposed as atomic functions while WSDL extensions provide syntactic metadata.

Compliant services are modeled using the descriptive logic capabilities of the Ontology Language for the Web (OWL). The resulting geospatial service model has a number of functions. It provides a basic taxonomy of geospatial Web services that is useful for templating service compositions. It also contains the necessary annotations to allow discovery of services. Importantly, the model defines a number of logical relationships between its internal concepts which allow inconsistency detection for the model as a whole and for individual service instances as they are added to the catalog. These logical relationships have the additional benefit of supporting automatic classification of geospatial services individuals when they are added to the service catalog.

The geospatial service catalog is backed by the descriptive logic model. It supports queries which are more complex that those available using standard relational data models, such as the capability to query using concept hierarchies. An example orchestration system demonstrates the use of the geospatial service catalog for query evaluation in an automatic orchestration system (both fully and semi-automatic orchestration).

Computational complexity analysis and experimental performance analysis identify potential performance problems in the geospatial service catalog. Solutions to these performance issues are

presented in the form of partitioning service instance realization, low cost pre-filtering of service instances, and pre-processing realization.

The resulting model and catalog provide an architecture to support automatic orchestration capable of complementing the multiple service composition algorithms that currently exist. Importantly, the geospatial service model and catalog go beyond simply supporting orchestration systems. By providing a general solution to the modeling and discovery of geospatial Web services they are useful in any geospastial Web service enterprise.

Computer Science, Web services, Orchestration, Ontology, Geospatial

Chapter 1

Introduction

In the last few years there has been a significant push by the US Navy, as well as other government agencies, to move its information technology infrastructure towards Web services. Geospatial Web services form a major component of this movement, because of the importance of oceanographic, meteorological, and navigational data and processing for the US Navy. Our primary work has been to research and design geospatial Web service architectures for this data and processing, the ultimate goal being to simplify the creation of existing and new geospatial products. That means simplifying data discovery, including both raw data such as survey results and derived data such as map images. It also means simplifying data processing tasks such as running models or interpolating grids. Final products are generally made from a number of different data sources and processing tasks. Web services are a perfect fit for the needs in this domain, but it must be possible for non-technical users to easily use the resulting Web service architecture.

Of course, simply creating a large number of geospatial Web services does not lead to widespread data sharing and distributed processing. New problems arise with the adoption of these technologies, notably implementation differences between internal groups, difficulty in discovering services, and the inability of target users to apply these services to their full potential. A key goal in the application of Web services is the ability to combine them, creating composition with more advanced functionality. The process of creating these compositions is called Web service or-

chestration. Traditionally, service orchestration has been a complicated manual process requiring significant technical knowledge of Web services.

Automatic orchestration of Web services is an area of research with the goal of reducing the complexity of service orchestration by automating portions of the process such as service discovery and service composition. Because of the complexity of manual Web service orchestration there is significant demand for some type of automated system. However, the challenging nature of the problem means that no fully automated Web service orchestration method currently exists.

In general, automatic Web service orchestration research focuses primarily on business services. To date, a limited amount of research exists in the realm of automatic orchestration for geospatial Web services. Geospatial Web services are designed especially to support geospatial data and processing and have become popular within scientific, military, and consumer systems. Geospatial Web services are designed around the movement of data, either for the purposes of supplying it or processing it. Business services, and therefore most automatic orchestration research, are focused primarily on the external real-world effects performed by Web services. Formally, geospatial services are stateless, whereas business services are stateful. As a result, much of the existing research in automatic orchestration does not provide optimal solutions for geospatial Web services. Service annotation, discovery, and composition all require substantial research to create a functional automatic orchestration system for geospatial Web services.

Our work focuses on a topic with little formal research in either geospatial or business Web services: modeling and cataloging Web services with goal of enabling automatic orchestration. Effective modeling of the service domain is a necessary first step to overcoming the inherent complexities in creating an orchestration system. The ontology is the semantic model we define to provide a richer description of individual geospatial Web services and their relationship to each other. Adding semantics to Web services is a useful way of describing services beyond the syntactic capability provided by a WSDL or other XML descriptions. Our goal was to create an ontology which is descriptive enough to back a geospatial service catalog targeted toward Web service orchestration.

A service catalog is a queryable database of Web services. Such a catalog designed for automatic orchestration must have a number of specialized properties. First, the catalog must store service annotations based on the semantic geospatial service model. Second, the catalog must be able to service Web service orchestration queries. These are queries designed to discover services which will fit into a specific position of a Web service composition. Lastly, the service catalog must efficiently evaluate these queries. An orchestration process will perform a large number of queries to create a composition. Query evaluation must support the performance requirements of an orchestration system.

The responsibilities of the catalog and its backing domain model make it a central component of a geospatial Web service orchestration system. It is also a component mostly ignored by existing research, especially in the area of geospatial Web services. Creating a geospatial service and catalog which can successfully model the variety of services in the geospatial area and perform queries over those services is key to creating a truly integrated geospatial Web service architecture.

This work is meant to achieve the following goals:

- Support Open Geospatial Consortium services in a traditional Web service architecture.
- Create a geospatial Web service model which:
  - provides a taxonomy of geospatial Web services.
  - provides the necessary annotations to support discovery of services.
  - supports automatic classification of services and inconsistency detection.
- Create a geospatial Web service catalog which:
  - supports user queries based on theme, producer, data type, etc.
  - supports orchestration queries based on matching service inputs and outputs.
  - provide update and query performance requisite with the needs of a functional catalog as part of a Web service architecture.

The outline of this dissertation is as follows. Chapters 2 and 3 provide background on Web services and automatic orchestration. Chapter 4 presents a survey of the related literature. Chapter 5 presents a method for supporting geospatial Web services based on Open Geospatial Consortium standards in a tranditional Web services architecture. Chapter 6 discusses the design of the geospatial Web service model. Chapter 7 presents our design and implementation of a geospatial Web service catalog backed by the geospatial Web service model. Chapter 8 presents our analysis of the system's performance and presents implementation choices which improve this performance. Chapter 9 discusses our conclusions and possibilities for future work in this area.

Chapter 2

Web Services Background

Web services are a general class of software systems designed for the purpose of supporting machine-to-machine interaction over a network. The goal of Web services is to have systems available on the Internet providing and processing data or performing business transactions. Web services allow previously disconnected software systems, both within an enterprise and between enterprises, to communicate with one another. Using Web services, complex interaction that once took substantial human involvement to move information between groups can now occur with no intervention from a person [5].

To accommodate use of Web services by remote parties, each service should provide machine-processable messages and interface specification. Machine-processable messages and interfaces allow machine-to-machine interaction between a Web service and its client. Additionally, a machine-processable interface allows automatic generation of the interface portions of the client, i.e., the sections of the client that actually connect to the Web service. Once the client and server are created, no further intervention by a person should be necessary [15].

While not required, there are two technologies which are commonly used components of Web services: HTTP (Hypertext Transfer Protocol) and XML (Extensible Markup Language). HTTP and XML are important to Web services because they simplify the goals of machine-to-machine interaction over a network. HTTP is the protocol used by the World Wide Web. The common use of HTTP is the impetus behind the term "Web" services. HTTP is an application-level, stateless,

client-server protocol. Because of its extensive use for Web browsing, HTTP is well supported by network and software infrastructure. The existing high level of support for HTTP allows Web service developers to concentrate on the unique capabilities of the service, rather than software tools, server and client software, custom protocols, network filters, etc. Web services may use alternate application-level protocols, such as the email protocol SMTP; however, in practice this is rarely done [14] [5] [15].

XML is the most common data encoding for Web services. While the general concept of a Web service does not require the use of XML, most Web service standards mandate its use in some form. XML is a specification for a customizable markup language for encoding data. It is designed to be easily readable by both computers and humans. XML does not place limitations on the structure or content of data. However, groups may create XML document types which do have these restrictions. These standards serve to simplify data transfer between applications. XML is commonly used in Web services both for its ease of processing and its ability to be easily restricted for particular application domains [6].

2.1 Geospatial Web Services

Geospatial Web services are a subset of Web services with a focus on geospatial data and processing. These services are focused on tasks such as providing imagery and vector data, geocoding, weather forecasting, route planning, etc. Geospatial services have the same underlying design as other Web services; they are of particular interest because they are generally more complex that other Web services. Geospatial Web services wrap processes that are highly complex. For example, a geospatial service which provides map imagery must specify the area where imagery is available, the allowable projections and datums of the imagery, supported image formats, available map styles, maximum image size, etc. The input parameters to specify these properties are complex objects. Additionally, the imagery returned may be quite large and the process to create it computationally costly.

Geospatial processes are important because their complexity shows the true strength of Web services. By implementing these processes as Web services the amount of costly reimplementation of functionality is reduced. Removing the need for in-person communication between processes increases the response time and reduces error. For example, a common means of distributing map imagery has previously been physically sending media to users. Mailing media is both costly and time consuming, as well as prone to error when a user must manually determine which CD or DVD has geographic area matching their request or decide if the map projection will work within their system. Web services reduce these costs, in both time and money, and automate difficult choices or delegate them to domain experts.

Geospatial Web services are significantly different from business Web services. The complexity of business services is derived from their effect on the real world. These services are used to sell books, ship packages, or book an airline flight. In order to properly model business services, these stateful effects on the real world must be included [19]. In contrast, geospatial Web services are stateless. These services either create or transform data. Modeling a geospatial Web service must primarly focus on data effects rather than changes in the real world. Rather than the complexity of state-fullness, geospatial services have complexity in data. Geospatial services use and produce data products in a myriad of different formats, data types, and with complex metadata. Any system which uses geospatial services must model this data complexity. As a result, solutions for modeling and using geospatial Web services must use different approaches from those for business Web services.

2.2 W3C Web Services

The most popular Web service standards are those created by the World Wide Web Consortium (W3C). The two primary standards are SOAP (not an acronym), a data exchange protocol, and the Web Service Description Language (WSDL), the method of specifying the interface to a service. SOAP and WSDL are the core components of the many Web service standards defined by the W3C. This collection of standards will be termed W3C Services for simplicity.

The W3C Service standards specify only the data exchange interface and the specification language for services. The W3C does not require the presence of any functionality in a service. A W3C Service may do whatever its creator wants, however, it must use SOAP as a data exchange protocol and specify its interface in a WSDL.

The flexibility of functionality in W3C Services allows these services to be used for any purpose. However, the flexibility also add complexity when using these services. There is no standard method of specifying what a service actually does. The WSDL only provides the function names and arguments. No exact description of what the function actually does is included [17]. A service may provide satellite imagery over a specific geographic area, but there is no way to adds this functional restriction to the service description. Data inputs may only be specified by type and name. There may be ambiguities about what data is expected by the function. For example, a service may expect an image as input but may only specify base64 encoded binary data as a function argument.

The generality of W3C Services forces client applications to be created specifically to work with a given service. In the case where an application must access a small number of services known ahead of time this is acceptable. However, if the desire is to automatically discover and use a service matching a particular request, there are significant hurdles to overcome.

2.3 OGC Web Services

The generality of W3C Services is a problem when users desire a standard set of functionality from Web services. In the geospatial domain, there are several common classes of services. If these services each had standardized functionality, they would be easier to integrate into systems, and thus, more useful. The Open Geospatial Consortium (OGC) defines a number of geospatial services which have both a standardized interface and standardized functionality. By standardizing functionality, the clients for these services may be generic. Unlike the case of, W3C Services, there is no need to create a client for each specific service. As long as a services follows the OGC standard, the client which also follows the standard will work with it [28].

The OGC has created a large number of geospatial Web service standards. However, three are the most commonly used. The Web Mapping Service (WMS) is used to transfer georeferenced images from the server to the client. The Web Feature Service (WFS) is used to transfer vector data (points, lines, polygons, etc.) encoded using Geography Markup Language (GML), a geospatial specific XML subset. The Web Coverage Service (WCS) is used to transfer geospatial multidimensional raster data. As opposed to the WMS standard, the focus of WCS is on data encoded in formats not supported by Web browsers. WCS originally only supported grid data formats such as GeoTIFF or NetCDF; however, current versions of the standard allow any encoding format for data transfers [10][46][13].

The standards created by the OGC are popular because they are designed specifically for geospatial services. Not only is the functionality of these services well defined, but the standards also include syntax for specifying metadata about services. The metadata is important because it defined the appropriate domain and range for inputs and outputs of services. Without the metadata these services would be difficult or impossible to use. By including metadata and functional specifications in the standards, the OGC Services have gained popularity over the more general W3C Services for geospatial services and applications.

2.4 Orchestration of Web Services

Web services are similar to reusable code libraries in a standalone computing model. Libraries allow functions to be created and made available for wide reuse, without the users needing to know or understand the internal operation of the function. Similarly, Web service functions may be connected together to form larger systems which can perform more complex tasks.

For example, several Web services may provide image processing functionality such as resizing, transparency filtering, or histogram color correction. A user may want to use more than one of these Web services to process an image, such as resize and then transparency filter an image. To accomplish this the user would send an image to the resize service, then forward the result to the

transparency filter service. The chaining of these two services is called an orchestration or service workflow.

An orchestration is an executable process made up of Web services, where the execution flow control is specified by the orchestration creator. In concrete terms, an orchestration connects the data flow between services, so that one service's output is directed into another service's input. Creating an orchestration requires the user to choose specific Web services, decide how they should connect to each other, and specify the order in which they execute (which could be concurrently) [2].

Any programming language which supports calling Web services can be used to create orchestrations. The programming language's standard flow control mechanisms will provide the necessary functionality to control Web service execution sequences. The benefit of using programming tools to create orchestrations requires no more Web service infrastructure than what is necessary to invoke a single service at a time. There are two problems with this method of creating orchestrations. The first problem is that the process of creating an orchestration in this manner has no standardization. Each set of tools operates differently and they are not necessarily easily deployed between systems. For example, if an orchestration is created using Sun's Java tools there will be difficulty using it on a system dependent on Microsoft's .NET tools. The second problem is that these programming tools require a trained programmer. Often the target user of Web services is not a programmer, but rather a domain expert. This user could be a business manager or a geospatial analyst. A user of this type will often not know how to use software tools for developing with Web services.

Business Process Execution Language (BPEL) is a programming language designed specifically for defining service orchestrations. BPEL has constructs defined in the language to invoke Web services and coordinate their execution. It also contains flow control functionality common to most programming languages. The language itself is written using XML. It is common for BPEL programs to be created from within a graphical development environment which automatically creates the BPEL XML for the user. These environments provide a drag-and-drop interface for

connecting services together which allows the user to create orchestrations as if they were drawing flow charts. The graphical interfaces allows for faster development than writing BPEL XML directly, and makes Web service composition accessible to users with less programming experience [2].

Web service orchestrations are run using orchestration engines. An orchestration engine executes a Web services composition. The goal of BPEL is to have a standardized language for defining orchestrations of Web services which is independent of a particular orchestration engine. Most orchestration engines support BPEL to some extent, though full support of the latest BPEL standard with no proprietary additions is often lacking [32].

Orchestration of Web services still presents many difficulties, in spite of the existence of the BPEL orchestration language and the tools surrounding it. These include finding Web services, ensuring they are compatible with each other, and composing them to produce the desired result. The creators of orchestrations still need expertise in programming and the application domain of the Web services. The existing orchestration tools do not fully address these issues. As a result, users who desire to create orchestrations must have a thorough understanding of Web services, a significant limitation.

Chapter 3

Automatic Orchestration of Web Services

3.1 Geospatial Web Service Orchestration Example

Prior to further discussion of Web services, automatic orchestration, and our proposed work on a geospatial service catalog, we will provide a motivating example for automatic orchestration of geospatial Web services. As mentioned above, geospatial services are stateless: they are defined by their inputs and outputs. Thus, a geospatial Web service orchestration must also be defined by its inputs and outputs.

For this example, the desired output of the orchestration is a hybrid street/imagery map. This map has a background of satellite or aerial imagery but contains drawn streets and locations with identifying labels. Such a map is provided by the Google Maps Web application as shown in Figure 3.1.

In order to create a hybrid map, a user must specify input parameters. In this case there are two necessary input parameters: the geographic area the map should cover and the size of the output image containing the map. These input parameters and the desired output define the request to the automatic orchestration system.

The responsibility of the automatic orchestration system is to take this request and create a Web service orchestration out of it. The service orchestration is a collection of Web services which can be connected together into a workflow. When run, the workflow should create the desired output

Figure 3.1: Example hybrid map.

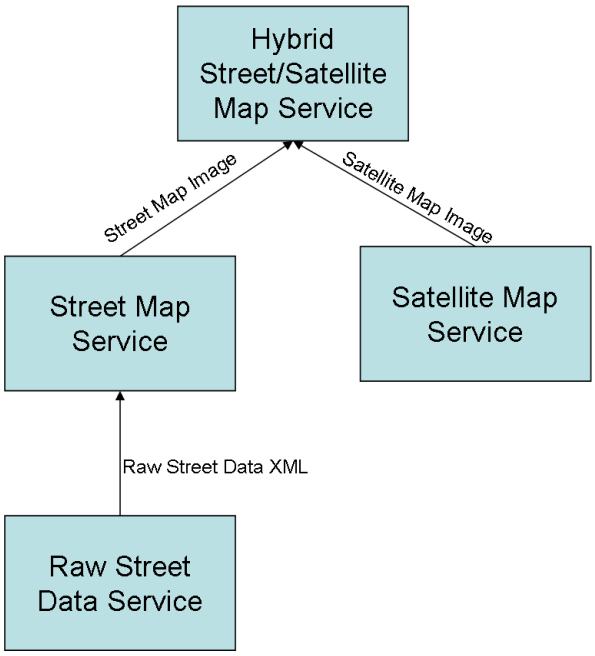


Figure 3.2: Diagram of the hybrid map orchestration.

using only the input parameters given by the user. For our example, the orchestration is a small workflow with four Web services as shown in Figure 3.2.

The resulting orchestration is based on a Web service which combines a satellite image and a street map. The satellite image service only requires the inputs from the user. In contrast, the street map service requires an additional input parameter to function: the raw street data. The street map service will take the raw street data and draw it to an image, but it must have another service provide it with the raw data. This separation of responsibility demonstrates one of the benefits of Web services. The functionality of drawing streets to a map is independent of the underlying street data. By separating the drawing from the source data, street data from any location can be used in the orchestration. The service providing the raw street data is able to run with only user inputs completing the orchestration. The complete orchestration can then be run by an orchestration engine which will take the user inputs and return the hybrid map image. From the user's perspective, the entire orchestration looks like a single Web service executing.

Automatic orchestration of Web services seeks to simplify the process of creating Web service compositions by automatically handling problems such as discovery of Web services, analysis of

their compatibility, and composition of service workflows. However, there is no accepted list of functionalities which an automatic orchestration system must have in order to be successful. Most work in automatic orchestration focuses on the composition task; however, there is significant additional functionality which an automatic orchestration must have in order to correctly create Web service workflows.

3.2 Annotation

The first problem to solve is for the automatic orchestration system to know exactly what individual Web services do. The orchestration system must know the function of a Web service as well as what constitutes valid input and output data. The current method of describing the capabilities of Web services, a WSDL, does not give enough information to satisfy this requirement. A WSDL provides a syntactic definition of a Web service. It defines the functions available in the service as well as basic type information of the input and output parameters of these functions. A WSDL does not provide any structured information about what a Web service function does or what the input parameters represent.

The syntactic function definitions in a WSDL give some information about input and output data requirements. However, in most cases the data types of inputs and outputs are not sufficient to completely know the data requirements of a Web service. For example, a Web service which does image processing may take a binary encoded image as input and return a modified image as output. However, any binary data would match the WSDL requirements for this service input parameter. For proper operation, the Web service requires the binary data to be an image, but the WSDL provides no means for the service to reveal this requirement. Similarly, there is no method of specifying that the output of this service is another image, rather than some random binary data.

For automatic orchestration to be possible, Web services inputs and outputs must be fully annotated. Data type and range restriction annotations are possible using existing Web service WSDL and XML schema definitions [17]. What is needed additionally is a means of defining the content, or the semantic definition, of the data. Semantic information about inputs and outputs is necessary

to determine service dependencies and possible compositions. While these types of annotations are not available in WSDL, they are available in the capability definitions of OGC Services. In the case of geospatial Web services, extending WSDL to allow the types of annotations available for OGC Services will be necessary to support automatic orchestration.

In addition to knowing the details of the inputs and outputs of a Web service, an automatic orchestration system must know what a service actually does. Currently, there is no information in a WSDL that describes the functionality of a Web service. There is an extension to WSDL, called SAWSDL, which adds semantic descriptions to the WSDL standard [21]. SAWSDL uses ontologies to provide the semantic information.

Most research into automatic orchestration of Web services use ontologies as the means of encoding the semantic description. Ontologies improve upon basic taxonomic classification schemes by also encoding the relationships between classes. Relationships between classes allow for more complex reasoning about entities based on the information in the ontology. Currently, the most commonly used ontology language is the Web Ontology Language (OWL) [27]. To successfully achieve automatic orchestration of Web services, an ontology must be created for the domain of services being orchestrated. For geospatial services, an ontology may be created using existing OGC and ISO standards as well as conventions common within the geospatial and environmental scientific communities. The domain ontology can be used in a semantic service description following the OWL-S standard. OWL-S is an ontology of services designed to enable automation in the use of Web services [26]. A primary difficulty in the automatic orchestration of Web services is the lack of any semantic annotations of services. Any automatic orchestration system must ensure that an ontology or other form of knowledge representation exists in the domain of the Web services being orchestrated.

3.3 Matching

Web service matching is the process of determining if the output of one Web service matches one or more inputs to another Web service, i.e., whether they can be "connected" together. If an

organization creates a collection of Web services it has the benefit of planning and control to ensure that the services are compatible and uniformly annotated. Data types used as inputs and outputs may share the same schema. Specific data types will always have the same format, encoding, and restrictions. For example, in such a controlled environment all services which use a geographic bounding box will encode it using the same XML schema type and semantic restrictions (i.e., only the EPSG 4326 projection). Unfortunately, Web services are rarely created in this manner. Web services are designed to be created by different entities with little central control. The lack of common standards for creating services leads to the requirement for Web service matching.

Many services are compatible even though they do not follow the same service creation standards. Simple translations or transformations of data would allow many services to be connected together. Determining whether services are compatible and performing the translations between them is the process of service matching. Determining whether services are connectible is difficult because compatibility must be determined at both a syntactic and semantic level.

Syntactic matching compares the data type compatibility of one service's outputs to another service's inputs. XML schema checking is the simplest method of performing syntactic matching. Unfortunately, this approach will create many false negatives.

To see this problem, imagine a Web service whose output is a geographic bounding box. The XML schema for the bounding box output is encoded as a BoundingBox XML document which contains four double precision numbers which are the minimum and maximum latitude and longitude coordinates of the bounding box. Suppose another Web service requires a bounding box as input. This input is encoded as a BoundingBox XML document which contains two GeographicPoints, each representing the minimum and maximum coordinates of the box. Each GeographicPoint contains two doubles which are the coordinate for the point. Syntactically the bounding box output of the first service is not compatible with the bounding box input of the second service. While the first service encodes the bounding box in a different manner from the second service, the actual data output by the first service is exactly what is required by the second service. Needed is

a method of reconciling the fact that the first service outputs the same four doubles that the second service requires as inputs, even though they are organized differently.

Semantic matching is necessary because syntactic matching is not reliable. Syntactic matching fails most easily with binary data. Whenever binary data is included in XML (a text based format) no information about the format of the binary data is included. Thus, syntactic XML matching is unable to differentiate between a JPEG image, a NetCDF data file, and a binary grid. Semantic reasoning about inputs and outputs is necessary to match properly.

Semantic matching of Web services depends heavily on the quality of the service annotation. Inputs, outputs, and functionality must be well annotated using an ontology or similar facility, as discussed earlier. The automatic orchestration system can determine semantic compatibility by reasoning about the semantic annotations of services. Semantic matching has the potential to completely determine if Web services are compatible. However, the ontological reasoning required for semantic matching is computationally expensive and becomes an untenable solution when large numbers of services are involved [36].

3.4 Discovery

The third important problem for automatic orchestration is the discovery of Web services. The orchestration system must know which services are available for use along with their requirements and functionality. A catalog of Web services is only useful if it is easy to find the correct services within it. As such, an effective Web service querying capability is an integral part of automatic orchestration.

Universal Description, Discovery and Integration (UDDI) is a catalog standard defined by OASIS [8]. However, UDDI is inadequate for the purposes of automatic orchestration. UDDI does not store any structured information about a Web service except its WSDL. As discussed earlier, a WSDL does not provide enough information about a service for automatic orchestration. Therefore, it is not possible for an automatic orchestration system to use UDDI as a service catalog.

A catalog usable for automatic orchestration must store all the annotations of a Web service required for later steps of the orchestration process. In addition to the WSDL, which contains syntactic information, the registry must store the semantic functionality of the Web service and metadata about the input and output parameters. These annotations are used for service matching. Retrieving services for service matching queries is the primary responsibility of the catalog in the automatic orchestration system. Additionally, the catalog must perform these queries quickly. Service queries will be performed constantly in an automatic orchestration system. Efficiency in computing these queries will be paramount to the successful operation of an automatic orchestration system. Thus, the primary goal of a service catalog in an automatic orchestration system will be to index services for efficient computation of service matching queries.

3.5 Composition

The composition stage of service orchestration is the process of connecting services together in order to fulfill an orchestration request. The composition process is the driver of the orchestration system. It uses the annotation system, the service registry, and service matching to create a final orchestration. However, the complexity of automatic orchestration is in these other tasks, not in composition.

Service composition is a basic search and graph creation problem. A successful orchestration is a directed, acyclic tree of services which, when connected together, satisfy the automatic orchestration request. The service composition task is to create the "best" tree as efficiently as possible. Which tree is the "best" can vary. It may be the tree with the fastest execution time or the tree with the least cost to run in terms of money or computational resources.

Composition may be done with a depth first search of the services to find matches dynamically. Alternatively, all matches may be determined ahead of time and stored in a graph which may be traversed to create compositions. Standard search and graph theory techniques such as these may be used. The challenge of composition is to determine which algorithm is most effective as the search space of services changes in size and/or complexity.

3.6 Execution and Validation

The final stage of Web service orchestration is the execution and validation of the generated service composition. Execution of Web service orchestrations is the process of actually using the orchestration to perform a function. The validation of orchestrations is the process of making sure the orchestration works correctly and the output is what the user intended.

For manual orchestrations of Web services both execution and validation are managed by software called an orchestration engine. An orchestration engine takes an orchestration specification as input, usually defined in BPEL or another orchestration language, and makes it available to users. Orchestrations in an engine look like standard Web services to an outside user. An orchestration will have a WSDL which defines the available functions. Web services clients will connect to the orchestration using SOAP as they would with any other service. The orchestration engine also monitors a running orchestration and detects faults as they occur, reporting them back to the user. These faults may be the result of an incorrectly created orchestration, a problem with an individual service, or a miscellaneous computer or network problem.

The functionality provided by the currently available orchestration engines is adequate for running manual service orchestrations. However, more functionalities would be beneficial for automatic service orchestrations. An automatic orchestration system can have the execution and validation of orchestrations tightly integrated into the entire orchestration creation process. The tight integration would allow the automatic orchestration system to handle faults in the system seamlessly. For example, if a Web service included in an automatically created orchestration is down, the automatic orchestration system could replace that service with another compatible one. A more tightly coupled orchestration engine would allow intelligent validation of outputs from an orchestration.

Chapter 4

Previous Work

Orchestration of Web services has been a popular research topic in the last decade as Web services and the semantic Web have gained in popularity. Geospatial Web service orchestration has received less attention; most research focuses on the business domain or specifies no domain at all. There are significant differences between geospatial and business services not addressed by existing research. In particular, business services are often stateful; they change the world in some way. On the other hand, geospatial services are stateless; they create or transform data. While other significant differences exist, the statefulness of business services has caused the existing Web service research into orchestration to be less useful for geospatial services.

There are three areas of automatic orchestration of services covered in the existing research. The first area is how to describe services so that automatic orchestration systems know what they do and how to use them. The section on ontologies (Section 4.1) discusses the current work in this area. The second area is in the indexing of services so that the orchestration system may find the correct services for a particular need. This work is discussed in the catalogs section (Section 4.2. The final area is service composition where the actual algorithms for combining services are discussed. This area is the focus of most research and is discussed in the composition section (Section 4.3).

4.1 Ontologies

For automatic, or even semi-automatic, orchestration to be successful there must be additional semantic descriptions of services beyond the interface description of a WSDL. The solution used in existing research is creating service descriptions using ontologies. An ontology is a knowledge representation system. An ontology usually contains a hierarchical set of classes and properties. The properties are used to define relationships between the classes.

4.1.1 DAML-S and OWL-S

DAML+OIL [16] and OWL [27] are both languages for representing ontologies. DAML+OIL is the older standard which has since been superseded by OWL. DAML-S [9] and OWL-S [26] are extensions of their respective languages which add classes and properties for describing services. Most research uses DAML-S or OWL-S as the means of representing their semantic annotations to services [23] [48] [7] [3] [45] [49] [24] [36] [35] [51] [22] [33] [31] [42] [29].

The limitation in the research that uses DAML-S or OWL-S is that they assume the ontology languages solve the service description problems of automatic orchestration. For example, Yau and Liu [48] use OWL-S as the basis for their service description and requests. However, there is no in-depth analysis of how ontologies can be properly incorporated into a Web service matching system or annotation of a non-trivial service. They only experiment with artificially generated services with OWL-S annotations autogenerated from a limited set of seed parameters. Thus, the services are easily matched and composed because they have none of the complexity and incompatibilities of real services [7].

This assumption that an ontological infrastructure exists which provides all annotations necessary for Web service orchestration is common among research in the field. Aydin et al. [3] use OWL for service description and only present a simple example for use in a “travel booking” problem. The assumption is made that the ontology exists and that all services fit within the existing ontology . Kalasapur et al. [18] claim ontologies as part of their orchcestration system but do not discuss which ontology language they use or provide any examples. Similarly, Le and He [23]

include OWL-S ontologies in their semi-automatic orchestration research but provide no discussion of how they should be used, concrete implementation, or examples. Yau and Liu [48] provide concrete services with ontology annotations in their examples, but use artificial services designed to work together easily within the same ontology. Liang and Lam [24] use OWL-S ontologies as a capabilities schema for their services, but do not ontological reasoning with the ontology data. Rather, they use simple keyword comparisons to determine similarity, defeating the purpose of using an ontology in the first place. Ren et al. [36] [35] also use OWL ontologies for service descriptions and then index the ontology using Quick Service Query List (QSQL) to improve the performance of service matching. They also do not provide a discussion of what should be included in an usable ontology of services, nor do they provide a significant example of a service ontology.

Timm and Gannod [45] present a "book search" example using OWL-S. They discuss encoding service connections and service parameters using OWL-S. However, the usefulness of their implementation is limited to an ontology and services created by one group. The services in their "book search" example are created to work with each other. Ontologies are most necessary in cases where the orchestration system itself must determine if the services are compatible and not just which service provides which function or require.

4.1.2 DIANE Service Descriptions

While DAML+OIL and OWL are the most commonly used ontology languages, an alternative language called DIANE Service Descriptions (DSD) was created specifically for describing services [19][22]. DSD is based on a light-weight ontology and was created to overcome shortcomings in OWL. The creators of DSD present some basic requirements for a semantic service description language. These include a functional description for service offers, functional description of service requests, domain-specific reasoning, an ontology language with specific elements for service description, and a shared ontology or support for ontology mediation. These requirements represent the best analysis of what is needed by a semantic description language in the current research.

The authors' analysis of OWL and OWL-S with respect to these requirements find it lacking, which is why they introduce DIANE Service Descriptions as a prototypical better system [22]. DSD is used for matchmaking in Küster et al. DSD [22], and their matchmaking work is focused on business services and is less useful for geospatial services. Business services are primarily defined by their effects on the real world such as purchasing an book, shipping a box of screws, or printing a document. DSD is designed to describe services using this effect-based model rather than one which focuses on a service's inputs and outputs. However, geospatial services must be described using inputs and outputs because they do not affect the real world. Thus, while DSD and related research represent the best analysis of service description languages, they do not meet the needs for geospatial Web services.

4.1.3 Geospatial Ontology

Efforts do exist to create geospatial ontologies for services but these focus primarily on adding semantics to data rather than the services themselves. A good example is the OGC Semantic Web Interoperability Experiment [25]. This experiment aimed to create a Geospatial Semantic Web architecture including a geospatial ontology for data and services. The goals of this work included adding semantic capability throughout the OGC architecture, not strictly limited to services. As a result, much of the work revolved around semantic additions for representing data, such as representing the Geographic Markup Language (GML) in OWL. An ontology based on GML does not help reach our goals for a geospatial service ontology, though it would complement it nicely. While the experiment does discuss using ontologies for services, results were limited to describing a WFS using OWL-S rather than the standard OGC XML schema.

Yaun et al. are currently working on creating a geospatial ontology. Their ongoing research is focused on taking the existing Geography Markup Language (GML) and converting it into a OWL ontology. As with the OGC Semantic Web Interoperability Experiment, the GML ontology approach does not meet the goals of a geospatial service ontology [49].

The COMPASS project from Stock et al. [44] is similarly broad-based as they present a semantic architecture for marine science data, services, and publications. A significant result of their work is an OWL application profile for the OGC Catalog Service for the Web (CSW) standard, providing a mechanism to store semantically annotated records for geospatial resources. However, the COMPASS ontology for the marine domain uses a theme categorization approach to semantics rather than a resource modeling approach.

Sheth et al., [39] present a Semantic Sensor Web architecture to provide enhanced descriptions and meaning to sensor data. Their work adds semantic annotations to sensor data for spatial, temporal, and thematic descriptions that improve upon the OGC standards for sensor services (such as the Sensor Observation Service). The results of the Semantic Sensor Web work provide better mechanisms to query and discovery sensor data, but the results are targeted toward sensor data discovery rather than service discovery.

The above work on geospatial ontologies, while presented in the context of Web services, is primarily data oriented. It focuses on the data provided by the services rather than the services themselves. As a result, it does not meet the needs of a geospatial Web service architecture designed for orchestration.

4.2 Catalogs

In addition to an improved service description language, service registries are necessary to successful service discovery. A list of services with their capabilities is a necessary prerequitsite to orchestration. However, research into catalogs for orchestration of Web services is limited. Much of the research does not directly discuss service catalogs or query evaluation [33] [37] [51]. Most existing research assumes the existence of a service catalog holding semantic metadata for all available Web services.

Orchestration research which does include a discussion of service catalogs falls into two groups: UDDI extensions and custom systems. UDDI extensions add support for semantic metadata to UDDI, the existing Web service catalog standard. Paolucci et al. [31] add a semantic catalog

layer above UDDI. All semantic search requests are handled by a DAML-S catalog, but standard keyword searches are available through the normal UDDI catalog. The semantic catalog will also automatically add UDDI catalog entries whenever it finds a new service to add to its semantic catalog.

Custom systems create their own catalogs not related to any existing standard. The VitaLab System created by Aiello et. al [1] is a service catalog designed for service orchestration. Syntactic similarity is determined by keyword comparison of WSDL "part" names. Semantic information is used only to create equivalencies between part names. The catalog index stores the part names and which services use them.

The catalog created by Yau and Liu [47] is limited to semantic matches of services. They compute matches based on "functional compatibility" which they define as the combination of parameter compatibility and conditional compatibility. Parameter compatibility determines if two Web services have semantically compatible inputs and outputs. Conditional compatibility determines if two services have compatible preconditions and effects.

QSQL is also semantic catalog which allows efficient reasoning over service ontologies. QSQL works by pre-reasoning on ontological service annotations and storing the results. Semantic service queries may be answered using information in QSQL instead of dynamically reasoning about services for each query [35] [36].

Similarly, in Sirin et al. [42] the orchestration system itself is a catalog which stores the ontological metadata about services. The knowledge base supports reasoning over the ontological metadata in order to support semantic queries. Their knowledge base differs from QSQL in that it does not pre-reason on service ontologies.

In Kalasapur et al. [18], the entire orchestration system is centered around the catalog. Their catalog stores all services in a large directed graph representing their composability; two services have an edge between them if the output of one may be an input to the other. The orchestration is done by querying the graph. The authors provide algorithms for both a centralized and a distributed catalog.

There are a number of limitations in this research using service catalogs. The majority of this work does not include an analysis of catalog performance, both for updates and queries. Automatic orchestration depends heavily on the query performance of the service catalog. Lack of query performance analysis means these solutions cannot be determined to be effective for automatic orchestration. Only VitaLab and QSQL include performance metrics in their research, but these are focused on the composition algorithm performance, not the catalog service. All of these research projects are primarily concerned with the composition problem and include a catalog as a component. As a result, the design and implementation of a service catalog in each project is not specified and difficult to judge.

4.3 Composition

The primary focus of most research into automatic orchestration is on service composition. The discussion of service descriptions and service catalogs in the existing research is usually subordinate to the goal of creating a composition algorithm. Web service composition takes a user request and finds one or more Web services which match the functionality in that request. In cases where services themselves have dependencies which are not directly fulfilled by user input parameters, further services may be found which provide the necessary inputs.

4.3.1 Matching

Web service matching is a limited form of Web service composition. The goal of service matching is to find one or more Web services which fulfill a request made by a user and have no dependencies beyond input parameters provided by the user. Web services are not connected to each other in Web service matching.

Paolucci et al. [31] present a simple matchmaking algorithm: check every service to see if the semantic annotations match the request. They provide the user with the ability to receive imperfect matches by using the hierarchy and relationships present in the ontology. The semi-automatic orchestration system created by Sirin et al. [42] is essentially a service matching algorithm assisting

users as they compose services manually. They discuss matching on functional and non-functional properties in the semantic descriptions of services but do not give details on what this actually entails. Liang and Lam [24] perform matching using probabilistic keyword comparisons rather than simple comparisons of the syntactic or semantic properties in WSDL or OWL-S documents. Their method matches even when parameters names are not exactly the same, improving on techniques where matching uses simple text comparisons. However, they do not use the semantic information of the OWL-S documents in their matching nor do they test with complex parameter types. Thus, their technique is more robust for existing simple services which suffer from lack of semantic information, but provide little help to the problem of matching complex geospatial processes.

Küster et al. [22] use DIANE Service Descriptions in their matchmaking system. Their matchmaking system is highly oriented towards business services which leads them to have a stateful model of Web services. As a result, user requests contain the desired effects of Web service rather than desired output. Also included are preferences which are used to limit Web service results and allow partial matching by the service matching system. The primary limitation of this work that it is restricted to matchmaking rather than generalized composition. Its focus on business services also limits its application to geospatial services.

4.3.2 Graph Theory

The result of a service composition is a directed acyclic graph. Each service in the composition is a vertex and the connections between one service's output to another's input is an edge. Since the result of service composition is a graph, it makes sense that a useful method of performing service composition is with graph theoretic algorithms.

Kalasapur et al. [18] create one large graph containing all services and then perform a path search to find service compositions. The actual algorithm uses two graphs, a semantic graph and a syntactic graph. The semantic graph connects services which are semantically compatible according to the ontologies, and the syntactic graph connects services which are syntactically compatible according to the input and output parameter definitions. The search algorithm walks the semantic

graph first, and then, using the results, does a path search on the syntactic graph to determine final compatibility. Kona et al. [20] also do a graph search but dynamically create the graph during the search. They query the catalog at each step to find services which match the query input parameters. They continue by adding services with the new outputs of the services already in the graph until they get to the query output parameters. The resulting graph is a conditional directed acyclic graph which includes information about pre/post conditions, information flow, and control flow allowing runtime changes to determine the solution path through the graph. Shiaa et al. [40] also dynamically create a graph when creating compositions. Their approach is more detailed and allows multiple goals within the composition, however, the overall approach is similar to other graph-based approaches. They use semantic similarity between inputs and outputs to connect services. In order to accommodate multiple goals, their technique connects multiple graphs together to create a larger composition which includes all the goals.

An analysis of performance is not provided by any of the above methods of graph-based composition. Theoretically, dynamically creating the graph during the search is better when the set of services is dynamic and when there are a lot of services. However, the performance of this algorithm can be poor if the service search is not optimized. More research should be done to determine when precomputing the graph is more efficient than dynamic graphic creation and search.

4.3.3 Petri-nets

Petri-nets are a common method of modeling distributed systems and have been applied to Web services and orchestration by a number of researchers [23] [29] [35] [52]. Petri-nets provide a more expressive way to represent Web services than as a node in a graph. However, the actual algorithms of composition are similar to those used by the graph models of Web services. Le and He [23] create the system of Petri Nets ahead of time from all the Web services in the repository. Specifically, they use Reasoning Petri Nets which include semantic information about the services in their transitions. Then reasoning over the Petri Nets is performed to solve the composition problem. The user inputs and goal requests guide which transitions are used in the Reasoning

Petri Nets. Both Ren et al. [35] and Zhovtobryukh [52] use a recursive algorithm for computing compositions. A Web service is found which fulfills every dependency in the user request. If those Web services themselves have dependencies, further services are discovered to fulfill them. The process repeats recursively.

The Petri-net representation of services adds to graph-based algorithms by improving the representation of dependencies and effects of the services. Whereas the graph models often use inputs and outputs to determine composability of services, the Petri-net models allow the non-visible effects of Web services to be included.

4.3.4 Planners

Artificial intelligence planners are designed to determine the optimal method of completing a set of tasks. Hierarchical task networks (HTN) are a type of AI planning which has been used to perform Web service composition. Web services are viewed as operations which may be used to complete a task definition provided by the user. Sirin et al. [43] use the HTN solver SHOP2 for Web service composition. The primary limitation of their technique is the need for a well-defined task definition as input to the HTN solver. In this case, the task definition is a composition template encoded using OWL-S. Thus, the HTN solver need only fill in the Web service operations to complete the necessary tasks defined in OWL-S. Paik and Maruyama [30] also use SHOP2 to solve HTN encoded Web service compositions, but combine it with a Constraint Satisfaction Problem (CSP) solver. The addition of the CSP solver provides additional functionality such as scheduling to the composition process. Chen et. al [7] use a combined Markov Model and HTN planning approach to service orchestration. Their improvement over other planning approaches is to find multiple possible compositions which can be used interchangeably depending on user preferences or dynamic network conditions.

The AI planner approach to Web service composition focus heavily on services which effect real-world schedules. These services, such as appointment services or travel reservation services, are sensitive to sequencing, concurrency, etc. These issues are less relevant for stateless geospatial

services. The need for task definitions as input to the planners also makes these algorithms less useful for geospatial services, especially since the geospatial services have complicated input and output parameters descriptions which would be difficult to encode in the task definitions

4.3.5 Formal Models

The formal model approach uses algebraic and theorem proving techniques to create Web service compositions. Salaun et al. [37] use the CCS process algebra to represent Web services. A process algebra is used to define the service with the basic operators and define how it changes when certain actions occur. Compositions are created by automatic reasoning over the statements of the process algebra.

Rao et al. [33]use linear logic as a formal model use in computing compositions. Unlike Salun et al. [37], they describe services using DAML-S and then translate the descriptions into linear logic. Service capabilities are turned into axioms of linear logic. A user request is translated into a theorem and an automatic theorem prover is used to determine if the request may be fulfilled [33]. Similarly, Aydin et. al [3] translate OWL-S service descriptions into an event calculus. An OWL-S generic composition is translated into event calculus. Then the process is broken into atomic components and an abductive theorem prover is used to match the components to service instances.

As with the matching system in Küster et al. [22], these formal model methods are most geared toward business services. The formal models manage the complexities of business services well, such as statefulness and hidden actions not represented by the outputs of the services. What these formal models lack is support for the more complicated service descriptions required of geospatial services, especially for input and output parameters.

4.4 Geospatial Orchestration

Currently, there is little research into the area of automatic orchestration of geospatial Web services. The research in the GeoBrain project provides the majority of the work on geospatial orchestration

[11] [12] [50]. The orchestration system created for the GeoBrain project is similar to some of the non-domain specific automatic orchestration systems discussed earlier.

Ontologies are an important part of the geospatial orchestration system of GeoBrain. They use semantic annotations to describe functionality and input/output parameters. The semantic annotations are encoded in DAML-S or OWL-S similar to other orchestration systems. The weakness of their approach is the limited discussion of what specific annotations are necessary for successful geospatial orchestration. No fully annotated service is provided in their work, nor do they present a significant ontology to be used for geospatial services. Reference is made to metadata standards such as Dublin Core and existing ontologies such as SWEET [34], but there is limited analysis of what these description standards provide, where they may be lacking, and how they are integrated into the orchestration system.

Their discussion of catalogs is similar. The OGC Catalog Service for the Web (CSW) standard is referenced; however, it is not capable of holding and searching the semantic descriptions of services. An extension which sits atop the CSW is described, but with no detail of storage, searching, indexing, or scalability.

The algorithm used for geospatial Web service composition is the recursive search algorithm used by many other orchestration systems [35] [52][51]. Little detail is presented on the recursive algorithm itself. Their discussion of composition is mainly focused on matching relations for services. Their system supports three types of matching: exact, subsume, and relaxed. Exact matching requires the request and response to be semantically equivalent. Subsume matching allows the response to be a subclass of the request. Relaxed matching allows the request to be a subclass of the response. Other semantic orchestration research has similar matching levels [35] [31] [36]. Unfortunately, there is little analysis of the error the expanding matching methods may create, and how this should be managed in an environment where orchestrations are created and executed automatically.

Implementation of a geospatial automatic orchestration system using the ideas of this research is either non-existent [11] or very limited [12] [50]. The most complete orchestration implemen-

tation described uses a simple problem with a limited number of services [12] [50]. Little detail is provided about the actual implementation and no analysis of the performance of the different components is provided.

Research into automatic orchestration of geospatial Web services must take into account the differences between geospatial services and business services. The Petri-net, AI planning, and formal model approaches to composition are improvements on more naive approaches to composition because they take into account the salient properties of business services. A well-defined model for geospatial services is necessary to create a complete geospatial orchestration system. Previous work on geospatial services has not created such a model. Our research aims to both design that model and implement a geospatial service catalog for use with it.

Chapter 5

OGC and W3C Web Service Interoperability

Architecture standards and technologies designed for Web services are built for the SOAP/WSDL standards of the W3C. Geospatial Web services most often use the standards defined by the OGC. A viable architecture for orchestration requires that OGC services work with the existing orchestration tools designed for W3C services. The success of enabling this will determine the viability of a geospatial orchestration system. We focus on a solution that uses a Web service wrapper around an OGC service. This wrapper provides all the functionality of the OGC service but with a standard W3C Web Service interface.

Data handling is an important issue to resolve between the Web service wrapper and the OGC service. Web services use XML as the primary method of communication. All non-string data types must be handled via special means. OGC services use a variety of data types not limited to XML. The three most common OGC services all request data using URL-encoded parameters in the HTTP request or in XML. Both methods translate easily into the Web service messaging model inside a simple wrapper. The difficulty arises in creating the response message. Each of the three OGC standards returns different data types. Though WFS uses XML, WMS and WCS use binary types. Because SOAP is strictly textual XML, these binary formats must be converted during transmission.

Another important consideration for the OGC to Web service translation is the mapping of functionality. Web services specify different requests as functions which are specified in the

WSDL. OGC services have one request to perform operations (getting/modifying/sending data) and other requests for metadata (capabilities document/type definitions). The operations of an OGC service are provided through one "function": GetMap for WMS, GetFeature for WFS, GetCoverage for WCS. The precise dataset provided by an OGC service is hidden within the Capabilities Document. There is no method of determining what data a service provides without calling the GetCapabilities function. How do we provide access to the functionality of the OGC service through the Web service, while also creating a usable Web service that follows best practices?

Metadata is one of the most important parts of the OGC service standards. Each standard requires geospatial metadata be added to the Capabilities Document. This metadata is crucial for its effective use by a client application. However, no standard for geospatial metadata exists for Web services. The Web service WSDL provides metadata in addition to function definitions. Removing the metadata from the Web Service would remove much of its usefulness. As a result, it is important to create an effective method of including metadata inside the WSDL.

5.1 Service Translation

The simplest method of data handling between the OGC and W3C services is to return only string data types from the Web service. Obviously, this presents a problem for OGC services that return images or binary files. One solution is to require that the client retrieve binary data from the OGC service directly. The Web service interprets the client's request and returns a URL-encoded request for the OGC service. We call this method "Service Translation" because there is no direct communication between the Web service and the OGC service.

Service Translation works by creating a translation Web service which is designed to create request URLs for OGC services from a SOAP request. The client creates the SOAP request for data and sends it to the Web service. The Web service parses the request and creates an equivalent OGC service request and encodes it into a URL. Then the URL is returned to the client in the SOAP response. The client must then use the URL to retrieve the data directly from the OGC service.

The primary benefit of the Service Translation is that the Web service does not have to manage messages containing binary data. Removing this functionality from the Web service reduces the cost and complexity of communication. The Web service does not have to act as a proxy for the OGC service data, removing the associated computational and network costs. However, the reduction in complexity and the load on the Web service are pushed to the client side. With Service Translation, the client must manage communication with both the Web service and the OGC service. While the client need not know the details of requesting data from an OGC service, it still must make a second data request. Fundamentally, this is antithetical to the operation of Web services. The goal of an OGC service to Web service mapping would be to remove as much complexity as possible from the client, a goal which loose coupling does not achieve.

In this type of system, a portion of the communication takes place outside of the Web service framework. Thus, if Web service specific extensions, such as WS-Security, are required for access, the translation prevents access to the service. The value of Web services comes from operating within the Web service framework and exploiting the functionality it provides. Bypassing this framework greatly diminishes this benefit. This last disadvantage is the main reason we did not use this method in our interoperability system.

5.2 Service Wrapping

In service wrapping, the data of the OGC service is retrieved by the Web service and then returned to the client. Service Wrapping will increase the load on the Web service system and introduce complexities; however, it is in most cases the appropriate method of creating a Web service interface to an OGC service. Using a service wrapper will make all interaction with the client completely Web service based. Thus, any specific Web service requirements, such as use of WS-Security or WS-Reliability, will be possible. Clients for the Web service can be made easily with existing tools and the system will mesh well in an existing Web services infrastructure. Because it meets our goal of hiding the OGC service from the client and allows the use of all W3C Web service extensions, we chose to focus our work on this method.

Service wrapping requires that the Web service be able to include binary data in its messages, specifically the response from a WMS or WCS server. The problem of including binary data with a SOAP response is not unique to geospatial service interoperability; thus there are some existing solutions for this problem. However, none of these solutions provides the ease of access to data made possible by accessing the OGC service directly.

Two methods that return data from a Web service are available. The first is to encode the binary data in a string and return it within the XML. Base-64 encoding allows any binary data to be representing using only ASCII characters. The Web service encodes the binary data from the OGC service using base-64 encoding and returns it to the client. The client then must decode the base-64 data before using it. The main benefit to base-64 encoding of binary data is that the resultant string can be easily embedded inside the SOAP response. Any Web service framework will be able to handle base-64 encoded binary data since it is functionally no different from standard string data. The problem with base-64 encoded data is that it is 33% larger than the original binary file. For large binary files often returned from OGC services (such as large map images or GeoTIFFs) the increase in size will be significant. Encoding and decoding the base-64 messages will be computationally costly, especially if the service is high volume. The decoding task may have to be manually performed on the client side. While not difficult, it would be preferable to have the binary file available in its original form immediately, and delegate the binary data handling to the Web services framework. Certain frameworks will automatically decode base-64 data, but this is not a designated standard.

Rather than encode the binary data as a string, it would be better to transmit the unmodified data. Since binary data cannot be embedded inside the XML document, the optimal solution is to attach it to the document and reference that attachment from within. SOAP with Attachments (SwA) is one method of sending binary data with a SOAP message. This method attaches data in the MIME format common in email. The attached binary file is then referenced from within the SOAP XML message. SwA allows binary data to be included unmodified with the XML while still maintaining a reference to it from within the actual XML. We dismissed SwA as a potential

solution for two primary reasons. First, an entire SOAP message must be scanned to retrieve attachments because MIME uses text strings to delineate boundaries between parts. Second, using MIME precludes using Web services extensions such as WS-Security, because MIME cannot be represented as an XML Infoset.

The problems of SwA led to the creation of the second method we used called Message Transmission Optimization Mechanism (MTOM). MTOM uses the XML Binary Optimized Packaging (XOP) standard to include binary data in a file. All binary data is encoded using base-64 and included in the XML file. MTOM will package that XML document within an XOP package. All base-64 encoded data is removed from the XML and optimized, i.e. converted back to its original binary form. The binary data is still attached using MIME but within an XML Infoset that allows Web services extensions such as WS-Security, which must compute signatures on the XML string data, to function properly. Using MIME allows MTOM to be backwards compatible with SOAP with attachments. MTOM retains compatibility with the Web services model because of the temporary state where the data is base-64 encoded. At that point all data is in string representation and usable by any extension or tool which requires compatibility with the Web services model. However, the transmission size is not inflated because the data is transmitted in the original binary format. The base-64 encoding of MTOM is also not a mandatory process; a client can access the original binary data from the message rather than having to base-64 decode it from a proper SOAP message. MTOM is a compromise which allows string-only representations of binary data without ever transferring the expanded form of the data. The main disadvantage of MTOM is that is must be supported by the Web services framework to be fully successful. Because MTOM is a relatively recent standard there are Web services frameworks which will not fully support it.

Our Web service interface to OGC services uses both base-64 encoding and MTOM as binary messaging methods. While base-64 encoding is not optimal, it will be supported by any Web services system. The encoding and decoding procedure is well known and easily implemented. MTOM is too new a standard to enforce its usage. However, the reduction in transmission size is useful for our system which is geared toward heavy usage. As a result, we implement both methods

of binary messaging in different functions, allowing the client to choose the method appropriate for their application.

5.3 Functional Mapping

W3C services and OGC services have a fundamentally different design. W3C services are designed to have a flexible set of functions which are described in the WSDL for the service. OGC services have a static set of functions but a flexible set of data. The data is described in the Capabilities Document for the service. There are two different ways to map functionality between OGC services and the W3C service wrapper, each useful in different contexts.

In the first method the WSDL of the W3C service lists the static function set of the OGC services. For example, the WMS specification defines a GetCapabilities and GetMap function, leading the W3C service wrapper to have corresponding GetCapabilities and GetMap functions. The client would be required to call the Web service version of the GetCapabilities function to fetch the metadata for the service and then call the GetMap function with the appropriate parameters. This is the method specified in the new OGC service standards. This method is appealing because it precisely matches the process of receiving data from an OGC service. Direct mapping is useful because it allows the same WSDL specifications to be used for all OGC services of the same type. It also allows the use of a Web service interface which is completely generic. Any provider of an OGC service can plug in a Web service interface following this model without any code or WSDL modification. The direct mapping also allows OGC services to be added to UDDI registries while still retaining the essential properties of the original OGC service. However, directly mapping functionality does not match the W3C Web service model. For a W3C Web service, all functionality should be revealed within the WSDL. By directly mapping OGC service functions into Web service functions all the important information about what the service actually does is hidden. For this reason, we use an alternative method to creating interfaces that better follow the Web service model.

Instead of a direct map between OGC service and W3C service functionality, we create a mapping between OGC service data and W3C service functionality. Rather than exposing a function, such as GetMap, the actual data layers are exposed, for example a layer such as "satellite imagery." The data layers can be exposed in two different ways. The first maps data layers of a single OGC service into functions in a new W3C service. The organization of the original OGC service is left intact and the relationships between the data layers are still apparent.

The second method maps each data layer into a separate service. This method creates a large number of simple, atomic services. No relationships or organization implied by the original OGC service exist within these separate services. With this approach, each new W3C service will have its own GetMap which returns the data from that particular layer. Each new service will have a W3C service port type containing GetMap function and can be easily composed into W3C service orchestrations with a Business Process Execution Language (BPEL) engine.

We chose the latter method of transforming OGC Layers into individual W3C services. To accomplish this, we created a tool which automatically transforms an OGC service into many W3C services. First, a parser ingests the OGC service capabilities document and determines the available layers. For each layer, the tool creates a WSDL that contains a GetMap function. This function will be used to retrieve the map data from the W3C service. The inputs of the GetMap function are the same as the original OGC service function except for the layer name which is hard-coded to the W3C service. Existing automated tools (WSDL2Java) are used to create the W3C service program code from the WSDL. The tool modifies this code to use an OGC service wrapper library which performs the actual request translation and forwarding to the OGC service. This system is completely automated, allowing the new W3C services to be created without any intervention. The process is similar for the alternate method of creating one W3C service with many functions except that only one WSDL is made with a function for each data layer. Metadata for each new service can be provided by an additional GetCapabilities function which forwards a portion of the OGC service capabilities document; however, we use an alternate method discussed in a later section.

As an example consider an OGC service with three map layers: "RoadMap," "SatelliteImagery," and "HybridMap." Our automated tool will find the three layers and create three separate WSDL files. The three WSDL documents will describe three new W3C services: "RoadMapservice," "SatelliteImageryService," and "HybridMapService". Each of these services will have a GetMap function with input parameters of geographic bounds, image size, etc. The WSDL documents will be used to create program code for the W3C services. The code for the GetMap function is modified to use the OGC service wrapper which will provide the actual responses to W3C service requests. In the case of the "RoadMapService" all requests will be turned into WMS URLs with the layername parameter set to "RoadMap." All other parameters are contained in the W3C request and are encoded into the URL. The URL will retrieve an image which is then sent back through the W3C service either base64-encoded or using MTOM. The newly created WSDL, auto-generated service code, and OGC service wrapper library can then be deployed and used via any standard W3C service mechanism.

5.4 Metadata

Metadata is important to the proper usage of any OGC service. The Capabilities Document of an OGC service contains the list of data sets available from the service as well as the geospatial parameters over which the data is defined. A particular map may be available only over a small portion of the globe. It may also be available in multiple spatial reference systems. Knowing these parameters is necessary to determine whether a certain piece of data is useful for a particular application. While the metadata specification is standardized in the OGC service Capabilities Document, there is no standard method of providing it from a Web service.

The Web service can provide access to the Capabilities Document of the OGC service. Any client may then use the data as if it had been obtained directly from the OGC service. But the Capabilities Document does not serve as the primary description document for Web services. As such, all Web service functionality based upon the use of WSDLs will be missing vital information

about the OGC service. Moving metadata into the WSDL will provide Web service-based tools and services with full information about a wrapped OGC service.

The problem is that a WSDL normally does not contain metadata beyond the functions provided by the service and the parameter/return types of those functions. WSDLs are extensible. The metadata for the OGC service can be included in the WSDL, but not in a standard way. While there are a limitless number of ways to encode metadata in a WSDL there are a few goals that should be met.

The first is that the metadata should not interfere with the proper usage of the WSDL. Any tools which do not use the metadata should not be affected negatively by it. Secondly, metadata methods should support all OGC service metadata and be consistent between different service specifications. And finally, the metadata encoding method should allow for simple validation of parameters using existing XML tools.

Our method for providing metadata encodes it inside the extensible portions of the WSDL. Extensible information is only allowed in certain portions of the WSDL. We include it inside the <service> element. The limits on the input parameters are encoded using an XML Schema definition with the XML Schema <restriction> element. Each input parameter is given a schema type which then has restrictions specified. For example, we can specify that the latitude of the request must be between 30 and 40 degrees or that the width of the return image must be less than 1000 pixels. The use of XML Schema to define capabilities provides a simple method of checking input parameters to the service. The input parameters can be validated against the restriction schema using a standard XML validation tool.

We add the metadata specifications for the response to its schema. Metadata elements are then placed within the response element in the WSDL. For example, the function may only return JPEG images. To signify this we add a format element with the string "jpg" to the response element. A client can parse these metadata parameters to determine the capabilities of the Web service. Our method meets the three goals for encoding metadata in a WSDL. Information is only placed in the extensible portions of the WSDL, thereby not interfering with usage of the WSDL with any Web

```
<Layer>
<Name>RoadMap</Name>
<Title>Road Map</Title>
<SRS>EPSG:4326</SRS>
<LatLonBoundingBox SRS="EPSG:4326" minx="-180.0" miny="-90.0"
maxx="180.0" maxy="90.0"/>
<BoundingBox SRS="EPSG:4326" minx="-180.0" miny="-90.0" maxx="180.0" maxy="90.0"/>
</Layer>
```

Listing 5.1: Geospatial metadata from a Capabilities Document

service tool. The XML schema can encode all input restrictions defined in OGC services and also allow simple validation of input parameters.

This method allows us to capture the information from a GetCapabilities document in the service WSDL. By encoding metadata in the WSDL, we preserve the OGC two-step process of getting a service's capabilities and then executing the service. Consider the example layer shown in Listing 5.1 from a WMS Capabilities Document.

The XML Schema snippet in Listing 5.2 defines an element "RoadMapRequest" and encodes restrictions on the values of the input parameters. These elements represent the definition of the input parameters to a SOAP service and would be encoded in the WSDL or a referenced schema document.

An alternate solution would be to use WS-MetadataExchange. This would allow metadata about the service to be encoded as WS-Policy documents. While this adds a third step back to the process, it does follow a W3C standard methodology. However, this standard has not been widely adopted so its use does not outweigh the implementation costs.

```
<xsd:element name="RoadMapRequest">
  <xsd:complexType>
    <xsd:sequence>
<xsd:element  name="bbox"                          type="geotypes:BoundingBox"/>
      <xsd:element  name="size" type="geotypes:Size"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="BoundingBox">
  <xsd:sequence>
    <xsd:element name="latMin" type="xsd:double"/>
    <xsd:element name="latMax" type="xsd:double"/>
    <xsd:element name="lngMin" type="xsd:double"/>
    <xsd:element name="lngMax" type="xsd:double"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="specification">
    <xsd:complexType>
        <xsd:all>
  <xsd:element name="latMin" type="latRestriction"  minOccurs="0"/>
  <xsd:element name="latMax" type="latRestriction"  minOccurs="0"/>
  <xsd:element name="lngMin" type="lngRestriction"  minOccurs="0"/>
  <xsd:element name="lngMax" type="lngRestriction"  minOccurs="0"/>
       </xsd:all>
     </xsd:complexType>
</xsd:element>

<xsd:simpleType name="latRestriction">
      <xsd:restriction base="xsd:double">
  <xsd:minInclusive value="-90.0"/>
  <xsd:maxInclusive value="90.0"/>
       </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="lngRestriction">
       <xsd:restriction base="xsd:double">
   <xsd:minInclusive value="-180.0"/>
   <xsd:maxInclusive value="180.0"/>
       </xsd:restriction>
</xsd:simpleType>
```

Listing 5.2: Geospatial metadata converted to a schema form to embed in a WSDL.

Chapter 6

Geospatial Web Service Model

6.1 Goals of a Geospatial Service Model

Often there is an assumption that adding semantics to a Web services means annotating it with a theme or functional description that is missing from a WSDL (which only provides function names, parameter names, and parameter types). And while thematic and functional descriptions are a component of a semantic model, we actually intend the semantic model to be much broader than that. A geospatial service semantic model should provide a formal representation of our application domain, including type definitions, encodings, functionality, etc. As discussed by Singh and Huhns [41], a model should facilitate reuse of resources within an enterprise, support resource sharing between enterprises, and allow validation resources as well as the model itself.

To properly build an ontology for an application domain we have to define exactly the functionality desired by our architecture and how the ontology will achieve that goal. The primary goal of our architecture is to support user or machine discovery and composition of geospatial Web services. To achieve this goal we need a number of capabilities for geospatial services:

1. Search for a service based on its functionality or theme.

2. Search for a service by input or output parameter type, encoding, or other internal property.

3. Find a service to produce an input for a target geospatial service.

4. Simplify the process of annotating a service semantically.

5. Validate service definitions.

These last two items are extremely important. One of the common barriers to implementing an architecture such as this one is the focus on how all the services will be properly annotated with metadata (semantic metadata in our case). Designing the ontology with simplified or automated annotation and validation in mind will reduce the hesitance to accept the architecture because of annotation concerns.

With these goals in mind we create the following requirements for our geospatial service ontology:

1. Provide a basic taxonomy of geospatial Web services and related entities that are used within our architecture.

2. Provide the necessary annotations to allow discovery of services (theme, producer, encodings, etc.)

3. Support simplified or automatic classification of geospatial services and inconsistency detection.

The first requirement is simply that all the geospatial services we are interested in within our architecture be included in the ontology, along with other necessary related entities. This requirement is a moving target. Our ontology does not need to define all possible service types, but should be expandable to support new service types as necessary. The second requirement ensures the ontology can help answer the question "What does this service do?" The third is perhaps the most important requirement. It states that the ontology should provide logic which will both test for inconsistencies in geospatial service definitions or support automatic classification of services.

6.2 Geospatial Parameter Ontology

Our geospatial service ontology begins by modeling geospatial parameters, the inputs and outputs of services. We focus on the parameters of a geospatial service because we model geospatial

```
:GeospatialParameter rdf:type owl:Class .

:MapParameter rdf:type owl:Class ;
              rdfs:subClassOf :GeospatialParameter .

:CoverageParameter rdf:type owl:Class ;
             rdfs:subClassOf :GeospatialParameter .

:VectorParameter rdf:type owl:Class ;
               rdfs:subClassOf :GeospatialParameter .
```

Listing 6.1: Initial ontology for geospatial parameters.

services as stateless. The stateless nature of geospatial services means that we do not have to model the internals of a service. Instead, we are concerned with what data enters the service and what data leaves the service. All model logic in the ontology will be centered around the parameters associated with a service rather than the services itself. From the perspective of the ontology, two geospatial services are equivalent if they have the same input and output definition. An additional benefit is that the entire functionality of a service is encoded in the parameter. An output parameter has its entire semantic description associated directly with itself, without the need to know which service it originated in.

6.2.1 Parameter Type and Encoding

Our ontology for GeospatialParameter has three subclasses for each of the core geospatial types discussed above: MapParameter, VectorParameter, and CoverageParameter. We further subclass these to provide necessary subtypes. GridParameter is a subclass of CoverageParameter, and VectorParameter has three subclasses which limit to data to a single geometry (PointParameter, PolylineParameter, and PolygonParameter). These data types are a mechanism for us to conceptually partition geospatial data; they do not describe the way data is encoded for storage or transfer.

The ontology shown in Listing 6.1 defines the concept of a geospatial parameter and makes a basic type hierarchy for classifying these parameters. Our model ontology uses a decidable subset of OWL 2 and displayed using Notation3 encoding here.

In order to describe how data is actually stored and distributed we added the class GeospatialEncoding to the ontology. Describing the data encoding is necessary because data with the

```
# Encoding Classes
:GeospatialEncoding rdf:type owl:Class .
:CoverageEncoding rdf:type owl:Class ;
                  rdfs:subClassOf :GeospatialEncoding .
:GridEncoding rdf:type owl:Class ;
              rdfs:subClassOf :CoverageEncoding .
:MapEncoding rdf:type owl:Class ;
             rdfs:subClassOf :GeospatialEncoding .
:VectorEncoding rdf:type owl:Class ;
                rdfs:subClassOf :GeospatialEncoding .

# Encoding Individuals
:BMP rdf:type owl:NamedIndividual , :MapEncoding .
:GIF rdf:type owl:NamedIndividual , :MapEncoding .
:GML rdf:type owl:NamedIndividual , :VectorEncoding .
:GeoTIFF rdf:type owl:NamedIndividual , :GridEncoding .
:HDF rdf:type owl:NamedIndividual , :GridEncoding .
:JPEG rdf:type owl:NamedIndividual , :MapEncoding .
:JPEG2000 rdf:type owl:NamedIndividual , :MapEncoding .
:KML rdf:type owl:NamedIndividual , :VectorEncoding .
:NetCDF rdf:type owl:NamedIndividual , :GridEncoding .
:PNG rdf:type owl:NamedIndividual , :MapEncoding .
:Raw rdf:type owl:NamedIndividual , :CoverageEncoding .
:Shapefile rdf:type owl:NamedIndividual,:VectorEncoding.
:TIFF rdf:type owl:NamedIndividual , :MapEncoding .
```

Listing 6.2: Basic ontology for geospatial data encodings.

same conceptual type is often stored using one of many different encodings. For example, images may use any of the these encodings: JPEG, PNG, uncompressed TIFF, JPEG2000, GIF, BMP, etc. Listing 6.2 defines the encoding classes and the encoding individuals.

The addition of GeospatialEncoding to the ontology allowed us to begin defining relationships between entities in our model. For this, we defined a property called hasEncoding which defines a relationship between the class GeospatialParameter and the class GeospatialEncoding. Along with the new property we also add a number of equivalency restrictions on the different subclasses of GeospatialParameter. These restrictions link the parameters with their respective encodings. The ontology section in Listing 6.3 shows the definition of the hasEncoding property and the encoding restriction for MapParameter.

The hasEncoding restrictions we place on the different GeospatialParameter types are a good example of the model logic within the geospatial service ontology which separates it from a simple taxonomy. These restrictions ensure entities are correctly constructed. For example, an inconsistency will be flagged if a VectorParameter has a MapEncoding. The restrictions also allow automatic classification of GeospatialParameter individuals. An ontological reasoner knows that any

```
# hasEncoding property states that a GeospatialParameter has a
# GeospatialEncoding
:hasEncoding rdf:type owl:ObjectProperty ;
             rdfs:subPropertyOf owl:topObjectProperty ;
             rdfs:range :GeospatialEncoding ;
             rdfs:domain :GeospatialParameter .

# MapParameter is equivalent to the set of things with hasEncoding
# MapEncoding
:MapParameter owl:equivalentClass [ rdf:type owl:Restriction ;
                                    owl:onProperty :hasEncoding ;
                                    owl:someValuesFrom :MapEncoding
                                  ] .
```

Listing 6.3: Property and restrictions to allow reasoning on GeospatialEncoding and GeospatialParameter.

```
# Original individual declarations.  Notice they are only defined to
# be owl:Thing individuals, not a GeospatialParameter or subclass.

:LandsatService rdf:type owl:NamedIndividual ,
                owl:Thing ;
                :hasEncoding :JPEG .


# Inferred model.  Here each individual is classified into its proper
# type, GeospatialParameter and one of [ImageParameter, GridParameter,
# VectorParameter].
:LandsatService rdf:type owl:NamedIndividual ,
                owl:Thing ,
                :GeospatialParameter ,
                :ImageParameter ;
                :hasEncoding :JPEG .
```

Listing 6.4: Original and inferred ontologies for a three example geospatial parameters

GeospatialParameter (actually any Thing) with the property hasEncoding MapEncoding is a MapParameter. This restriction is a step toward achieving one of our goals: automatic classification of services and inconsistency detection. The ability of an ontology to contain this type of logic is a primary reason we are using one.

Listing 6.4 shows our model before and after reasoning to automatically classify parameters. Notice that the individual LandsatImageMap is only defined as class Thing. Only the encoding needs to be specified about the parameter and the reasoner automatically determines the parameter type. The reasoning itself is performed by the OWL 2 reasoning engine Pellet. However, the choice of reasoner is irrelevant. The logic itself is encoded in our model definition. Any standards compliant reasoner will create the same inferred model.

6.2.2 Data Theme

In addition to data type and encoding, we must also be able to characterize what a geospatial parameter actually represents. Is the map a chart or a satellite image? Does this grid represent sea surface temperature or bathymetry? By looking at the parameter defined in Listing 6.4 we may infer that LandsatService is a Landsat satellite image, but this cannot be determined by a computer. We must add this information to our ontology so that it is available to the knowledge system. We call the meaning of a parameter its theme. Parameter theme within our ontology is a taxonomy without additional relationships. We do not need DataTheme to have any properties on which to reason beyond simple hierarchy. It is tempting to add restrictions which limit themes to specific data types (such as Bathymetry limited to the Coverage datatype) but we do not want to limit the ways we can represent these different themes. Data usually represented as a Grid may be rendered into a Map, partitioned into Vector areas, or originally be collected as an irregular Coverage. The geospatial parameter themes for our architecture are presented in Figure 6.1. In addition to parameter themes, we have two more classes used to describe geospatial parameters: DataProducer and DataOrigin. DataProducer describes the organization which created the parameter and DataOrigin describes how it was created such as measured or estimated by a model. The ontology includes properties linking the GeospatialParameter class to the attribute classes DataTheme, DataProducer, and DataOrigin.

These attribute classes are necessary so that our ontology fulfills the first and second modeling goals discussed above. These classes fill out our domain vocabulary, allowing services to be fully described and those descriptions effectively shared between systems in the enterprise. These three classes are also key to supporting effective discovery of services. The theme, producer, and origin of data and services are frequently queried properties which would significantly limit the usefulness of our model if omitted. These properties will also be important as we define our different geospatial service classes and specify the model logic for them.

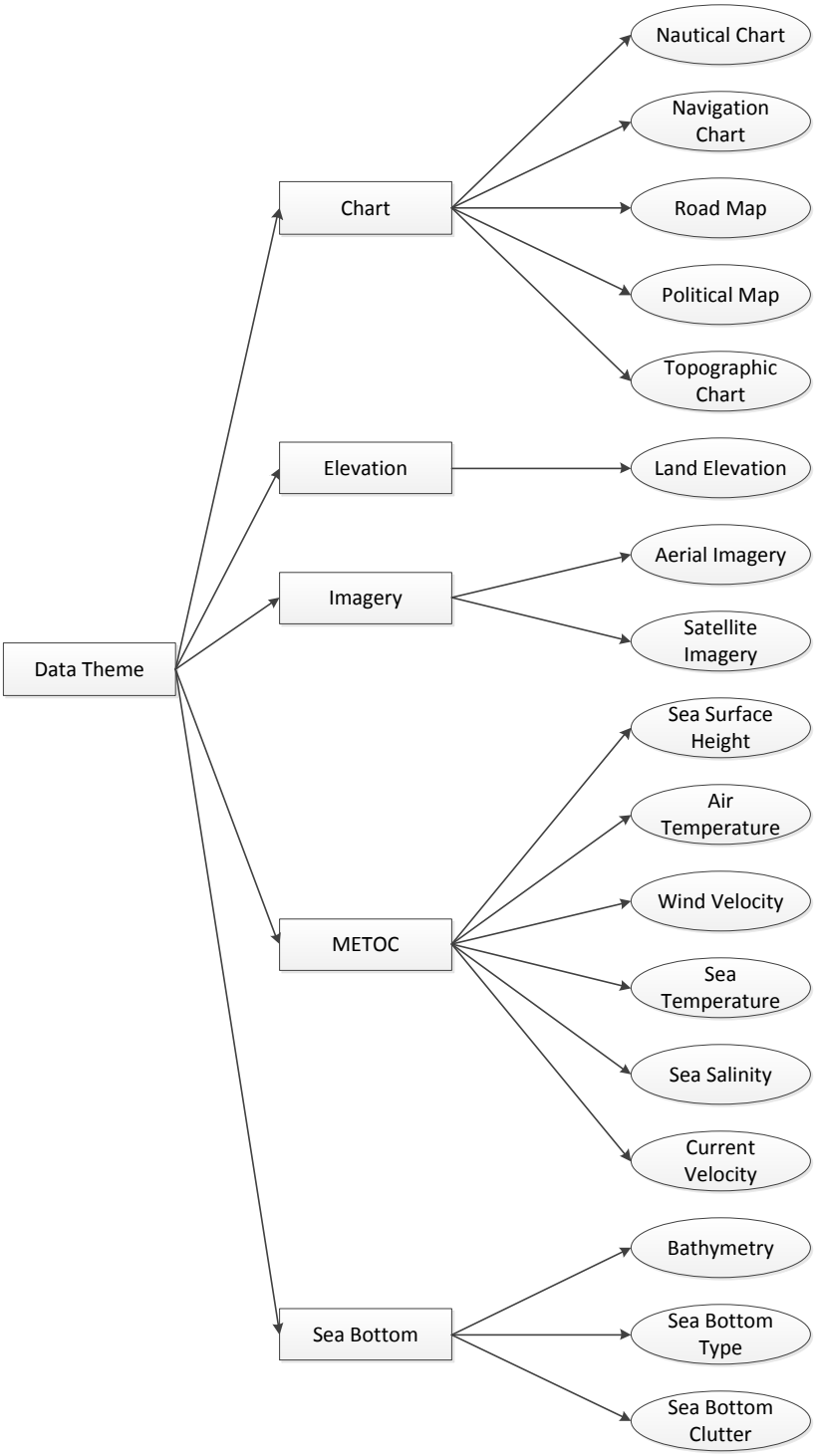


Figure 6.1: Our ontology includes classes for characterizing geospatial data themes. These are used to describe the purpose of a geospatial parameter.

6.2.3 Geospatial Parameters and User Parameters

So far we have limited our discussion of input and output parameters of a geospatial service to geospatial parameters: maps, vectors, and coverages. However, geospatial services require non-geospatial parameters to function as well. Consider a Web Map Service. Though this service takes no geospatial parameters as inputs, there are a number of non-geospatial parameters which are necessary to properly use this service, for example bounding box, image width, image height, etc.

These types of parameters function differently from the geospatial parameters we considered above. Geospatial parameters are the result of our geospatial services and form the data flow in a geospatial service composition. These non-geospatial parameters are not part of this data flow. Instead, they are defined by the user when invoking a service or an orchestration. As a result, we call these User Parameters to distinguish them from Geospatial Parameters. User parameters are analogous to arguments in many common line programs. While the primary data may be piped between programs, the arguments are always defined by the user. We include user parameters in our ontology but they do not form a part of the model logic that will affect reasoning and queries.

6.3 Geospatial Service Ontology

In our ontology geospatial services are defined by their inputs and outputs. We are able to do this because these services are stateless. Each different type of geospatial service is specified using its particular combination of inputs and outputs.

6.3.1 GeospatialService Definition

A GeospatialService is defined as WebService with a geospatial parameter as its output. Our ontology requires a geospatial service to have exactly one geospatial parameter as an output. We require at least one output because a stateless service with no outputs is useless. While a geospatial service could have more than one output parameter, the services in our architecture and most other deployed geospatial services only have one output parameter. (Note that a single vector output

```
# definition of a geospatial service
:GeospatialService rdf:type owl:Class ;

  # GeospatialService is a subclass of an anonymous
  # class
  rdfs:subClassOf [ rdf:type :WebService ;

  # the anonymous class is a intersection of
  # three restrictions
  owl:intersectionOf (


    # 1. the hasOutputParameter property always an
    # has object from the class GeospatialParameter
    [ rdf:type owl:Restriction ;
      owl:onProperty :hasOutputParameter ;
      owl:allValuesFrom :GeospatialParameter
    ]

    # 2. There is exactly one hasOutputParameter property
    [ rdf:type owl:Restriction ;
      owl:onProperty :hasOutputParameter ;
      owl:cardinality "1"^^xsd:nonNegativeInteger
    ]
  )
] .
```

Listing 6.5: GeospatialService definition for our ontology.

parameter is a collection of features rather than a single geometry). The inputs to a GeospatialService are a combination of geospatial parameters and user parameters. We place no cardinality limitations on input parameters, either geospatial or user parameters.

The GeospatialService class in our ontology is defined by its relationship to its input and output parameters. The hasInputParameter and hasOutputParameter properties link the GeospatialService class to the GeospatialParameter and UserParameter classes. The restrictions on these properties also ensure that a GeospatialService has only one output parameter of type GeospatialParameter. The OWL 2 definition of GeospatialService is shown in Listing 6.5.

6.3.2 GeospatialService Type Hierarchy

The basic GeospatialService ontology definition does not do much to allow us to classify services, support service queries, or create service compositions. For that we must add further specialization to the geospatial service definition. For this we classified the service types in our geospatial service architecture and added the types to the ontology as shown in Figure 6.2.

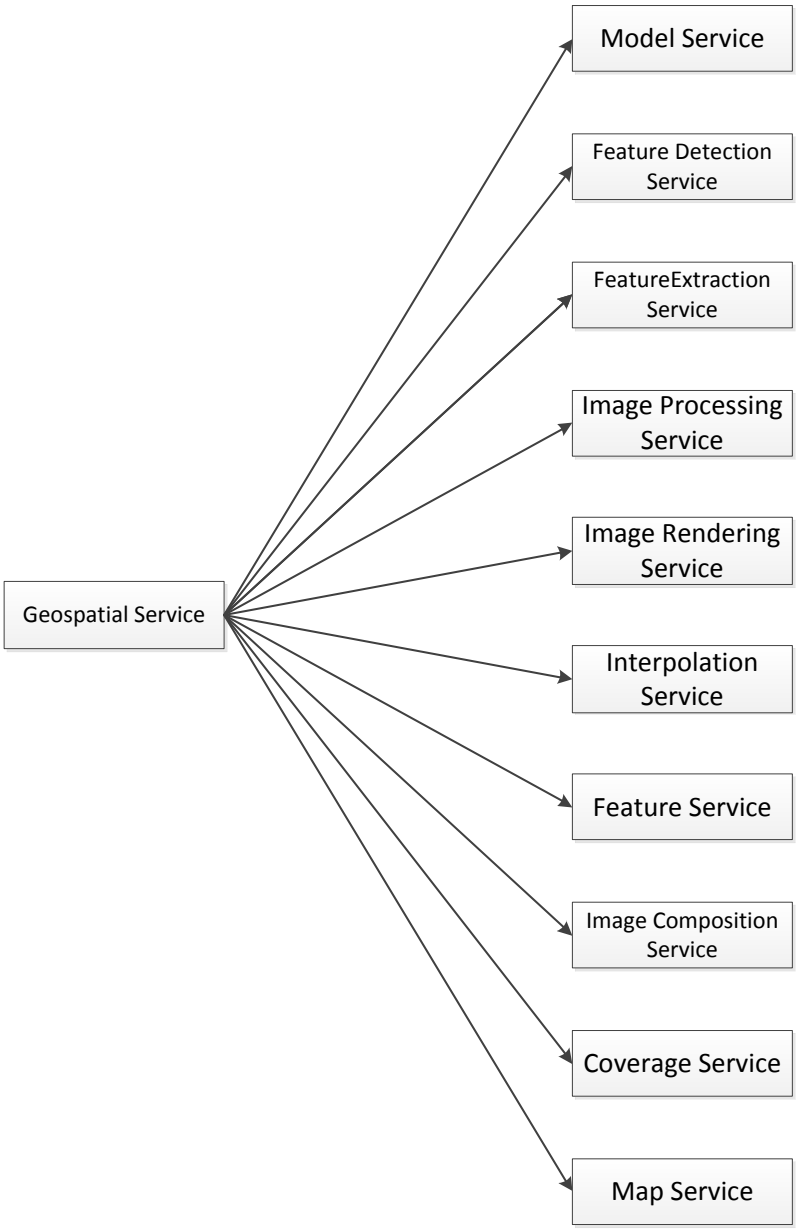


Figure 6.2: Ontology hierarchy for geospatial services.

Figure 6.3: The definition of the MapService is one of the simplest in our ontology. It defines a MapService as having a single ImageParameter output and no GeospatialParameter inputs.

If we had stopped building our ontology here it would be possible for us to manually classify each service in our architecture to be an individual instance of one of the above types. But without additional model logic we would not achieve the third goal for our ontology, automatic validation and classification. This is arguably the most important and the reason why we used an ontology instead of another mechanism for taxonomy definition (such as a basic XML Schema).

The first model logic we added to the GeospatialService classes was to restrict the number and type of their input and output parameters. Data creation services, such as MapService, are limited to zero GeospatialParameters as input. They each output a different type of GeospatialParameters: MapService is limited to ImageParameters, FeatureService is limited to VectorParameters, and CoverageService is limited to CoverageParameters. The definition of the MapService is shown in Figure 6.3.

The processing services are limited in the types of their input and output parameters. An ImageProcessingService is limited to one map input parameter and one map output parameter. An InterpolationService is limited to one input parameter, either a collection of PointVectorParameters or a CoverageParameter, and outputs a GridParameter. The other services shown in Figure 6.2 have their inputs and outputs defined similarly.

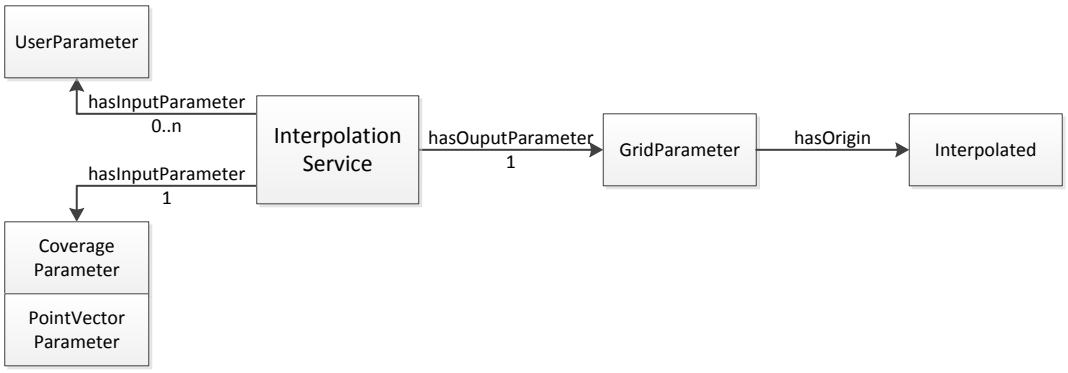
These initial definitions of our service types in the geospatial service ontology go a long way toward achieving our third goal. These basic data type restrictions allow us to categorize a large number of services. Along with the GeospatialParameter restrictions discussed earlier, it is possible

to perform reasoning to classify services with only knowledge of the input and output parameter data encodings for a service.

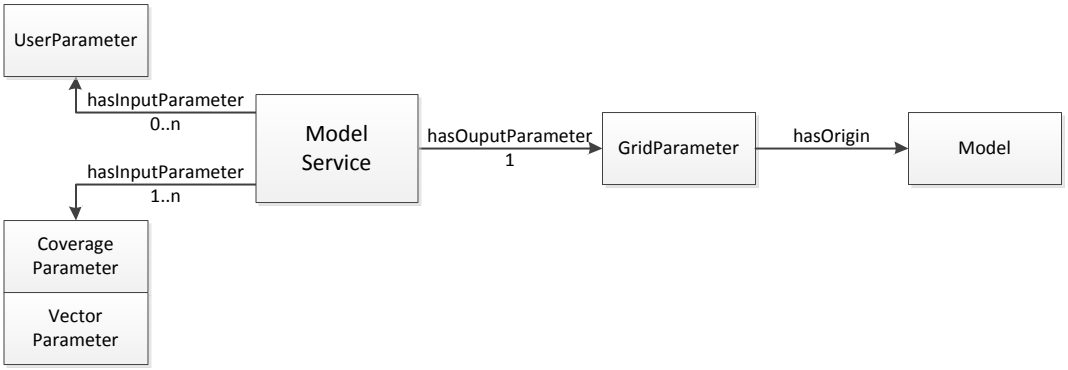
Additional ontology components are necessary to fully classify services and detect inconsistencies in the ontology and cataloged individuals. The InterpolationService and the ModelService demonstrate why we need more than just data type annotation. A ModelService takes an input of any type of raw data, vector data or coverage data. An InterpolationService takes similar data types: point vectors and coverages. If only the data type ontological annotation is used, the reasoner will determine that InterpolationService is a subclass of ModelService. Of course, in our architecture ModelService has a significantly different function compared to a InterpolationService. An InterpolationService creates a grid from its input. The output is fundamentally the same data as the input. On the other hand, a model uses its input to predict the state of an environmental phenomenon, resulting in a completely new dataset as output. The data type restrictions on services in our ontology cannot represent these differences.

Instead, we use the other properties associated with a geospatial parameter to further enhance our definitions and associated logic for the geospatial service types. To better define the ModelService we use the property hasOrigin to restrict its OutputParameter to having a DataOrigin of Model (either a Forecast, Nowcast, or Hindcast). On the other hand, for InterpolationService we restrict the OutputParameter to having a DataOrigin of Interpolated. This additional restriction on DataOrigin specifies to the reasoner the intrinsic difference between a ModelService and an InterpolationService. Now an InterpolationService is not reasoned to be a subclass of ModelService. Figure 6.4 show the differences between the two services.

All of the core service types in our architecture can be classified using data type and origin restrictions. But we also wanted to add further validation logic, beyond the restrictions defined for these core services. Further subclasses to our core service types are used to group services which share other properties beyond their input and output parameter data types. These subclasses add restrictions on the hasProducer, hasTheme, and hasOrigin properties. As an example, our service architecture contains a number of ModelServices all produced under the auspices of the U.S. Navy



(a) Interpolation Service



(b) Model Service

Figure 6.4: The definitions of InterpolationService and ModelService. The important difference between the two is the restriction on the OutputParameter. ModelService must have an output with an origin of Model and the InterpolationService must not.

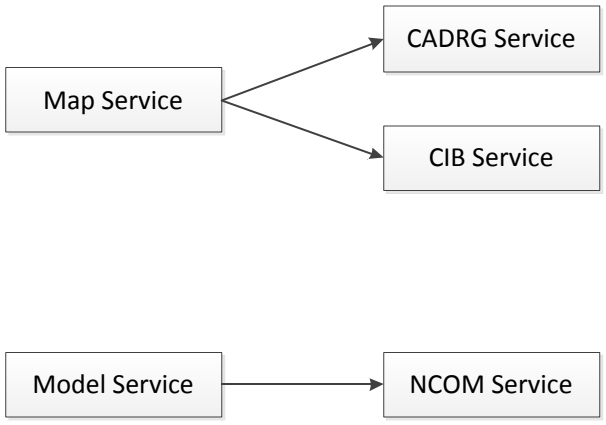


Figure 6.5: The geospatial servies which are defined as subtypes of our core services. These allow us to predefine specific property annotations.

Operational Global Ocean Model (NCOM). To support these services we created a subclass of ModelService called NCOMService (see Figure 6.6). The NCOMService has added properties on its output parameter. The DataProducer is restricted to the NAVO, the DataTheme is restricted to METOC, and the DataOrigin is restricted to either Nowcast or Forecast. The additional logic added to the NCOMService automatically adds the DataProducer information to the service if it is not already there. More importantly, these restrictions ensure that when a new NCOMService is defined it will be validated for proper metadata. Given that our architecture contains many groups of related services, creating service subclasses for them provides a mechanism to perform a wider array of validation checks. It also reduces the difficulty of annotating a service when a single choice of service type will indicate which metadata properties should be provided for the service and limits or removes the options for filling in those properties.

One of the benefits of creating the geospatial service model using an ontology is the ease with which it can be updated. This is best demonstrated with an example. Often it is necessary to find services which are publicly releasable to individuals outside our architecture. These are services which provide only US government civilian or community created data and do not require processing resources. First, we defined a type called ProcessingService which is a subclass of

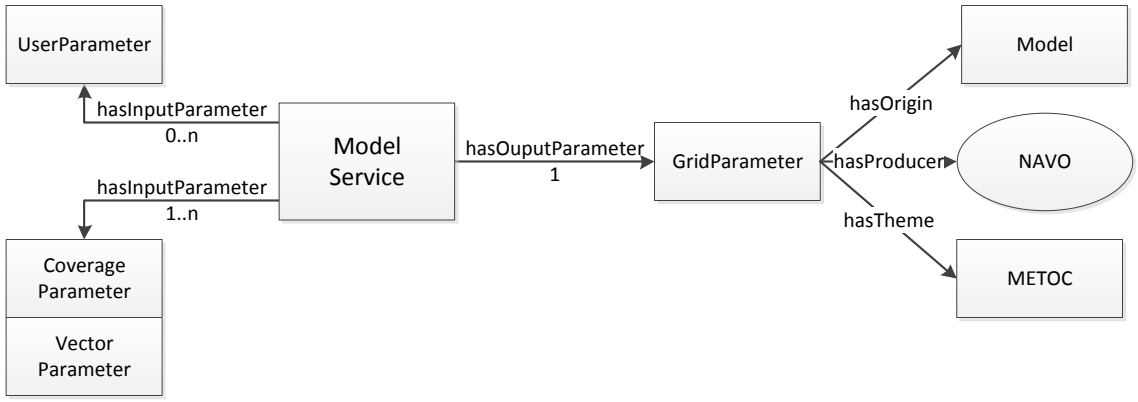


Figure 6.6: The restrictions on the NCOMService will automatically annotate DataProducer and provide validation for the other required properties.

WebService. A ProcessingService is defined as being equivalent to any service which takes a GeospatialParameter as input. It is important to note that the individual services defined in our catalog do not have to be re-annotated as being a ProcessingService. They are automatically classified as having type ProcessingService because they match the equivalence restriction defined for the ProcessingService type. The results of adding the ProcessingService type are shown in Figure 6.7. We similarly added a new type for PublicService which is defined as a WebService which is not a ProcessingService and has an output parameter produced by a civilian or community organization. The benefit of this type of update is that when the model is updated, all individuals are included automatically in the update. In comparison, a relational model would have to both update the model and run a complex custom update on all data to ensure proper annotation with this new information, or never update the schema and use a complex query to represent these properties. This update demonstrates how the built-in logic which comes from using an ontology can simplify the modeling process, especially as it evolves over time.

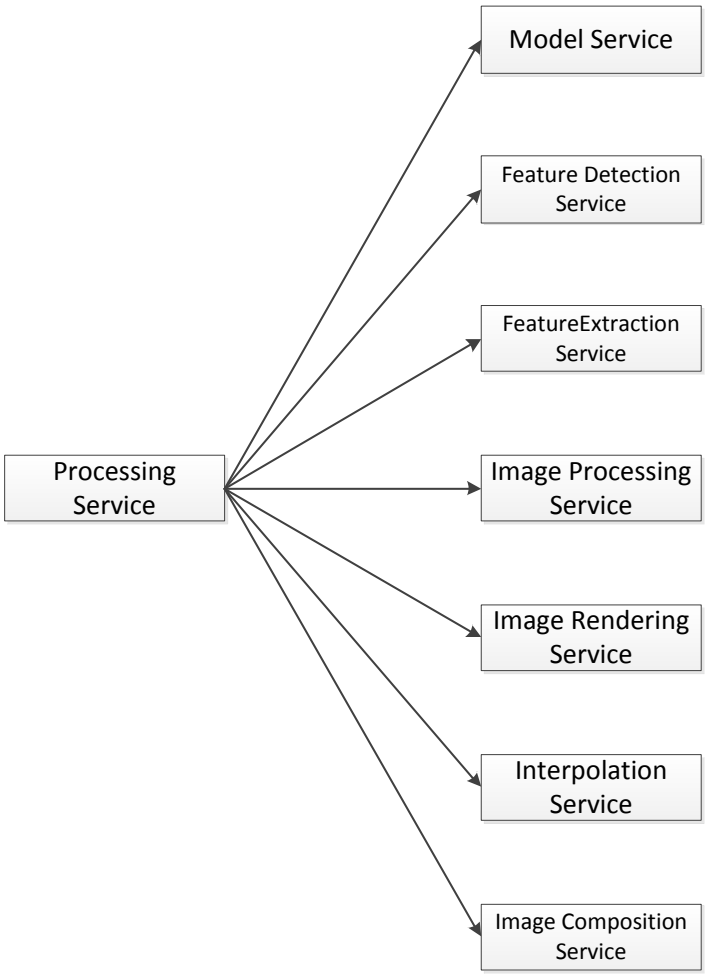


Figure 6.7: Resulting processing services after the ProcessingService class is defined in the ontology.

Chapter 7

Geospatial Service Catalog

The geospatial service catalog is the key architecture component which enables service discovery and, ultimately, provides the core functionality supporting automatic orchestration. A service catalog, backed by a detailed domain model, has two primary tasks: manage updates and evaluate queries. The service catalog maintains the complete listing of all available services along with the necessary metadata to discover and understand these services.

The catalog itself is based on our geospatial service model. The geospatial service model was designed specifically to support the update and query functionality of the catalog, but additional components must ride on top of the model in order to actually implement this functionality. One of the benefits of creating the model using the standards based OWL ontology language is that the catalog implementation can use existing software components to complete its functionality rather than implement these pieces from scratch.

7.1 Geospatial Service Catalog Design and Implementation

As stated earlier, the geospatial service catalog must support updates to its service listing and queries over those services. This data flow is shown in Figure 7.1. Of course, the detail of how these functions are provided is not clear from this basic data flow. Decisions were necessary on how data is stored in the catalog, how queries are specified, how queries are evaluated, etc.

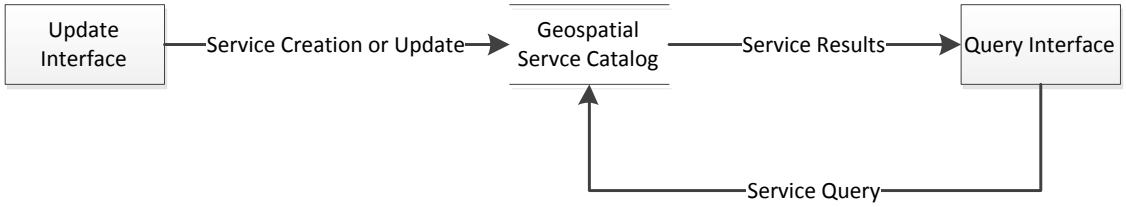


Figure 7.1: The basic data flow for the geospatial service catalog.

Many of these design decisions are made because of the requirement to incorporate the geospatial service model. The model, also called the core ontology, is defined using the OWL 2 ontology language. The core ontology is the schema for the data stored in the catalog. Thus, service definitions must also be specified using OWL. The service definitions are stored separately from the core ontology in what is called the individual set. Updates to the catalog require each new or updated individual be specified using OWL. Given the complexity of constructing OWL by hand, the catalog update user interface should provide a translation from a simple GUI input mechanism to an OWL statement. (While a GUI was not built for this work, a similar translation from a easy-to-create XML structure to OWL was implemented.) The OWL API (http://owlapi.sourceforge.net/), a Java API for creating and using OWL ontologies, is used to translate service definitions into OWL.

7.1.1 OWL, RDF, and Ontology Realization

The OWL ontology language is actually an extension of the Resource Description Framework (RDF). The RDF language encodes information using triples: statements in the form subject-predicate-object. The resulting set of triples forms a graph where subjects and objects are vertices and predicates are edges. Because OWL is built on RDF, all OWL statements are of this form and constitute a graph.

The difference between OWL and RDF is the inferencing capability designed into OWL. RDF is simply a language for encoding information, but OWL adds limitations upon how data is encoded, resulting in a functional description logic language. A key function available in a description logic is the ability to perform inferencing, i.e., deducing new statements from the set of existing logical axioms. This logical inferencing is also called realization of the ontology. The ability to perform realization is the primary reason the geospatial service model was created using OWL.

The realization process results in the creation of new ontology statements. These new statements are RDF triples just like the original OWL statements. These new statements most commonly specify class membership or add new object properties for an individual. Both the pre-realization set of RDF triples and the larger post-realization set are usable by any software component designed for RDF. All the realization process does is create that larger set of data upon which to run these components. In the geospatial service catalog, realization of the individuals set is performed by Pellet (http://clarkparsia.com/pellet/), an OWL 2 reasoner for Java.

7.1.2 Catalog Queries

Queries in the catalog are specified using SPARQL (SPARQL Protocol and RDF Query Language), a query language for RDF. SPARQL is designed to support queries over the triples statements of RDF. SPARQL is also the primary query language for OWL because OWL is built on top of RDF. The query interface should provide a simple user accessible method of constructing a query which is then translated into SPARQL. SPARQL query evaluation is performed by the Jena Semantic Framework (http://jena.sourceforge.net/). Figure 7.2 shows the overall catalog data flow with the query components visualized.

SPARQL evaluation does not handle an OWL graph any differently from a basic RDF graph. As a result, SPARQL evaluation over an unrealized set of OWL individuals will be “missing” some data. Instead, realization must be performed prior to SPARQL query evaluation. The exact

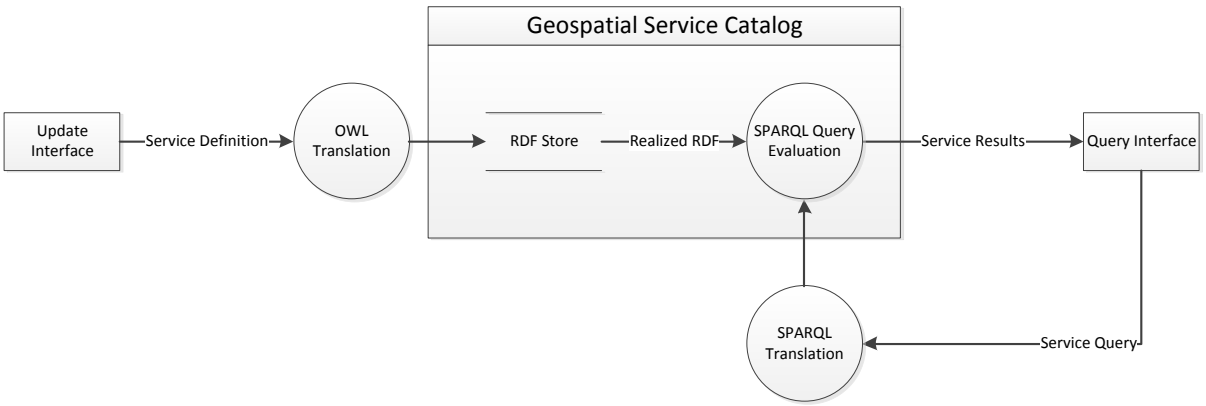


Figure 7.2: Detailed data flow for the geospatial service catalog.

location in the architecture where realization occurs is variable. The various options are analysed in the next chapter in conjunction with performance experiments that reveal their pros and cons.

7.1.3 Data Storage and Access

Data storage and access patterns for RDF data follow the existing patterns for other more common data types. Most commonly data is persisted with either file storage or databases. For database persistence, either a relational model or custom triple stores are used (Jena provides the option for either). The database option provides both long term persistence of RDF and an interface for direct queries of data. File storage is primarily used to persist smaller amounts of RDF or to share RDF data between systems. Flat files generally use plain text encodings of RDF. RDF/XML, N3, Turtle, and N-Triples are the most common encodings for RDF data. File storage does not provide direct queries of data. Instead, the entire RDF data set is loaded into memory before being queried.

The geospatial service catalog persists individuals in flat files and queries are performed using a copy of the data loaded into memory (shown in Figure 7.3). The decision to use flat files and memory for storage and access is based on the expected usage patterns of the service catalog. Each service definition is about five kilobytes stored in plain text using the N3/Turtle encoding. Memory consumption for a single service definition is comparable. One gigabyte will therefore

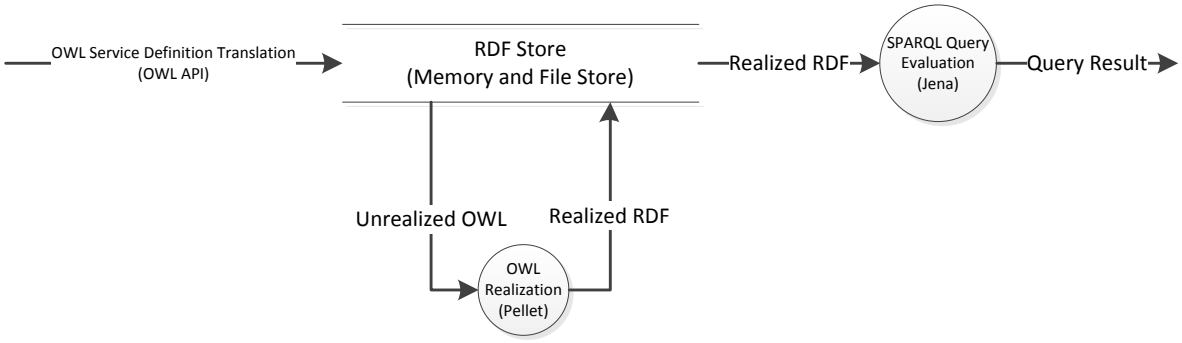


Figure 7.3: Internal data flow in the service catalog.

store about 200,000 service definitions. One gigabyte of disk storage is trivial and any system with the required processing power to host the service catalog will also have one gigabyte of memory easily available. Designing a system to accommodate 200,000 services vastly overestimates the number of services available in any current architecture, especially the architecture our geospatial service catalog is targeting.

Flat file storage also provides more flexibility for the catalog. The RDF file formats are standards (official or ad hoc) that are supported by multiple tools. In particular, the catalog uses both the OWL API and Jena to interact with the RDF data. Using the custom database formats from one of the APIs precludes usage by the other. Direct memory access of the entire set of individuals will also be faster than disk access through a database backend. Also, improved realization is possible by employing parallel realization of individuals (discussed in the next chapter). Being able to partition the individuals set between multiple flat files allows a simple and efficient mechanism for supporting concurrency that is not possible with a database backend. Lastly, storing all data in memory precludes the need for complex indexing schemes to improve database query performance. Adding an indexing requirement to the catalog would both over-complicate its design and reduce flexibility in the design of the realization, storage, and querying components of the catalog.

7.2 Geospatial Service Catalog Usage

The geospatial service model is specifically designed to support the operations of the geospatial service catalog. Evaluating the efficacy of the model requires testing it in the context of such a catalog. In particular, we are interested in three specific functions: updating the individuals or the model within the catalog, user query evaluation, and query evaluation for orchestration systems.

7.2.1 Updating the Catalog

One of the core goals of the geospatial service model is to improve the process of updating the catalog. First, the model must validate both itself and the individuals added to the catalog. Second, the model should classify individuals based on the known annotations, inferring as many unspecified properties as possible.

We have already seen in the previous chapter how classification is based on properties such as output parameter DataEncoding and DataOrigin. Equally important is the ability to detect errors in service definitions.

CADRGService and CIBService classes provide a good example for error checking. Our ontology defines an abstract superclass for both by requiring the data producer for these services be NGA. If a user adds a service marked as a CIBService or a CADRGService and has another producer, say USGS, then an inconstancy is generated when the updated ontology is realized. Similarly, if the theme of a CIBService is not imagery or the theme of a CADRGService is not a chart, an error is generated.

The first function of the geospatial service catalog is to support user queries for Web services focused on discovery of services to invoke manually. Second, the ontology-backed catalog supports assisted addition of geospatial services. As a key motivation in the design of the ontology, this capability expands upon a limited service description using the internal logic of the geospatial service model. It also supports consistency checking when adding new services. Lastly, the geospatial service catalog provides support of orchestration of services. For orchestration, rather than a user querying service, the orchestration environment queries the catalog, either to assist

users in generating a service composition or in checking a composition for potential errors. The key capability that an ontology-backed catalog provides is the ability to query based on derived parameters rather than only those manually entered into a catalog record.

The query language used by our catalog is SPARQL, a standardized query language for RDF graphs. The query functionality is made available via a service interface to user applications such as a web-based search page and service composition tools. Our system uses the Jena Semantic Web Framework as the query engine. The Pellet reasoner is used to create a fully realized RDF graph from our ontology which is then passed to the Jena SPARQL engine.

As discussed earlier, one of the primary goals for the geospatial service ontology is to support automatic classification of services and consistency checks for service definitions. The logic for providing automated checking and service classification is encoded in the ontology. Our ontology provides a mechanism to test for errors or provide automated update actions which is far more transparent and shareable than alternative methods such as database triggers. Encoding logic in an ontology abstracts model logic from backend system implementations.

7.2.2 User queries

Users query the service catalog by defining filters on the properties of the individuals stored in the catalog. A common user query is to search for services based on their theme, for example a query for services providing sea temperature. Through the query interface, the user makes a request for all services with the sea temperature theme. This query is translated into SPARQL for evaluation in the catalog. The theme property is attached to the output parameter of the service. In most cases, the queries in the service catalog will require filtering on the input or output parameter individuals rather than the service individuals alone. The SPARQL request for the sea temperature services results in a query similar to a join in the standard relational model. The query requests all services with an output parameter with the sea temperature theme. SPARQL queries are often similar to relational join queries because they must match patterns within the RDF graph. The

Query:	SELECT ?service
	WHERE
	{
	?service geo:hasOutputParameter ?output .
	?output geo:hasTheme geo:SeaTemperature .
	}

Result:	NCOMSeaTemperatureService NOAASeaTemperatureService

Table 7.1: A SPARQL query for services matching the sea temperature theme.

Query:	SELECT ?service
	WHERE
	{
	?service geo:hasOutputParameter ?output .
	?output geo:hasTheme ?theme .
	?theme rdf:type geo:METOC .
	}

Result:	NCOMSeaSurfaceHeightService NCOMSeaTemperatureService NCOMWindVelocityService NCOMSeaSalinityService NCOMCurrentVelocityService NOAASeaSurfaceHeightService NOAASeaTemperatureService NOAAWindVelocityService NOAASeaSalinityService NOAACurrentVelocityService

Table 7.2: A SPARQL query for services matching any METOC theme.

SPARQL query for sea temperature services and the result is shown in Table 7.1. The result is two sea temperature services: one NCOM service and one NOAA service.

The power of using SPARQL/RDF for the service catalog is revealed by a related query. Rather than search for a service with the sea temperature theme, the user chooses to query for all services with any METOC theme. While this query is similar to the previous one, it requires reasoning using the ontology type hierarchy rather than just querying directly annotated properties. Hierarchical evaluation is a capability enabled by using an ontology-based catalog rather than a relational-database driven catalog. The METOC theme query and its results are shown in Table 7.2. The results include all of the NCOM services in our architecture as well as the similar NOAA services.

7.2.3 Web Service Orchestration

The geospatial service catalog is designed to support automatic Web service orchestration. There are three types of automatic orchestration of interest: template-based orchestration, guided orchestration, and fully automatic orchestration. The first two types are semi-automatic orchestration methods which assist the user in creating a Web service composition. Fully automatic orchestration creates the entire composition with only a user provided goal. Semi-automatic orchestration is of the most interest in our architecture. Semi-automatic orchestration removes the need for advanced technical knowledge about Web services when making compositions but still allows fine grained control over what specific services are added to the composition. The lack of control implied by fully automatic orchestration makes it less attractive as part of a working architecture. As a result, the geospatial service catalog is targeted towards template and guided orchestration. The two semi-automatic orchestration methods initiate two different types of queries. The use of the catalog with these orchestration methods will be demonstrated separately.

Two example compositions are used to demonstrate the use the geospatial service catalog within an orchestration system. The first is the hybrid map service which overlays a rendered road data over a satellite image. Figure 7.4 is a diagram of the service composition for the hybrid map service. The second service is a landing plan service which combines multiple products into a unified landing plan product. The diagram of the landing plan composition is shown in Figure 7.5.

Template-based Orchestration

Template orchestration uses fully specified service queries. Requests are made for services using input and output parameter types, themes, origin, etc. The template defines a data flow similar to the diagrams shown in Figure 7.4 and Figure 7.5. A user is presented the data flow diagram without concrete services instances in place on each position in the template. The user then selects services to fill in each slot in the template.

At every step of a template-based orchestration the orchestration system presents the user with all possible services to fulfill a particular slot in the template. These possible services are deter-

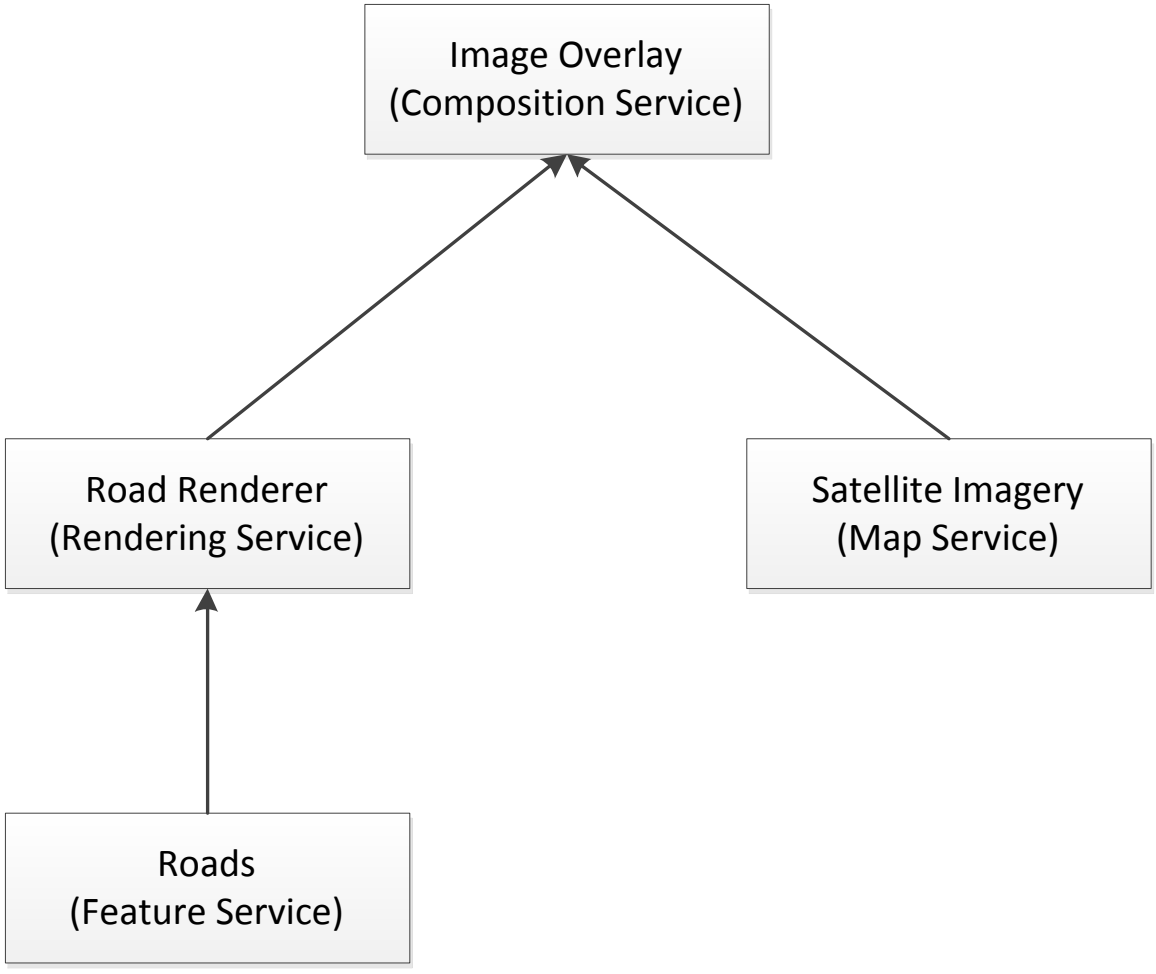


Figure 7.4: Hybrid map composition diagram.

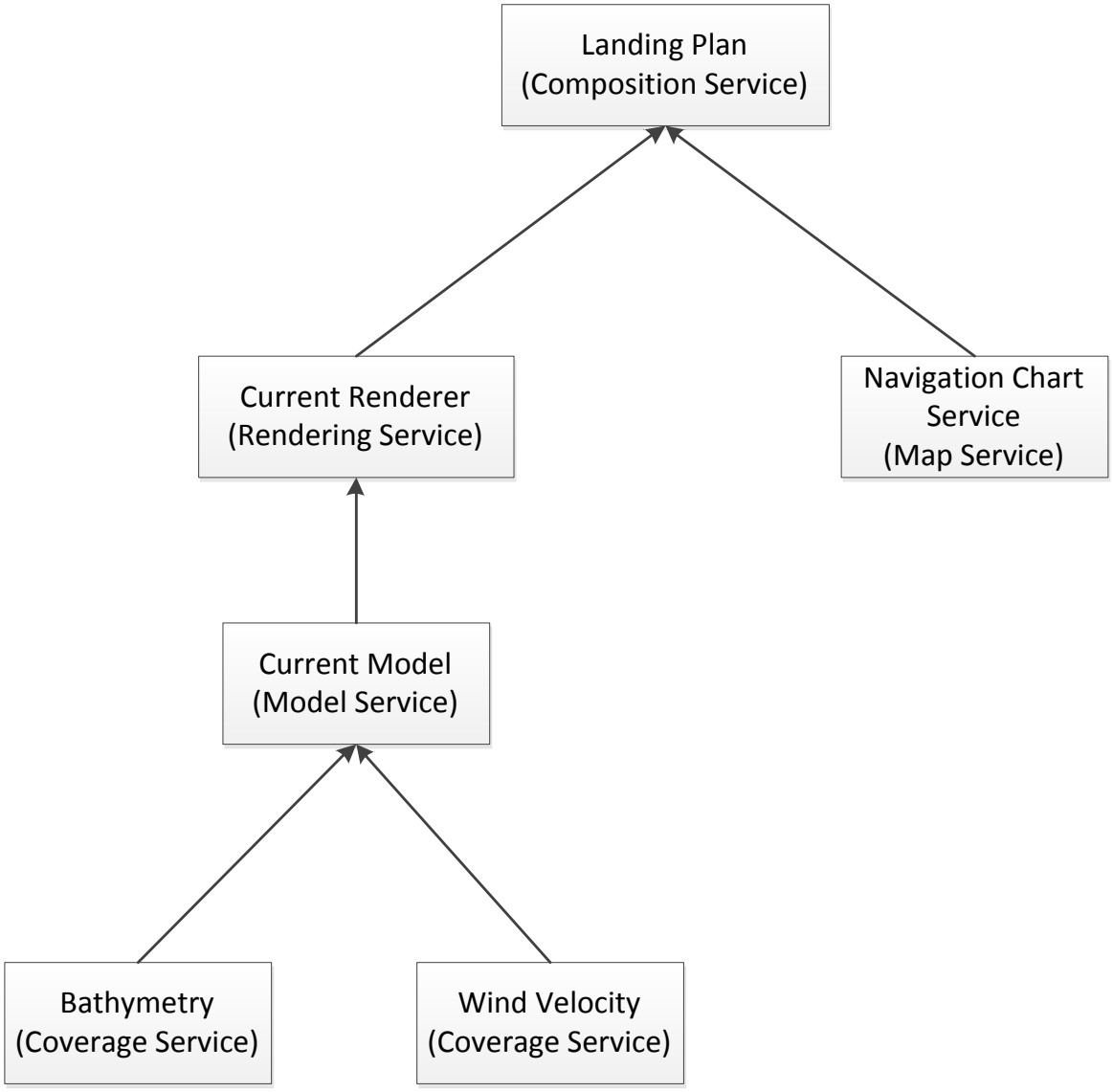


Figure 7.5: Landing plan composition diagram.

Query:	SELECT ?service
	WHERE
	{
	?service rdf:type geo:FeatureService .
	?service geo:hasOutputParameter ?output .
	?output geo:hasTheme geo:RoadMap .
	}

Result:	NavteqService OpenStreetMapService

Table 7.3: Template orchestration query for a road service.

Query:	SELECT ?service
	WHERE
	{
	?service rdf:type geo:RenderingService .
	?service geo:hasInputParameter ?input .
	?input geo:hasTheme geo:RoadMap .
	}

Result:	OSMStyleRendererService GoogleStyleRendererService

Table 7.4: Template orchestration query for a road rendering service.

mined by querying the catalog for services which match the specification for that particular service slot in the template. The queries required to fill the hybrid map template are shown in Tables 7.3 - 7.6. The results from each query is presented to the user who then chooses a specific service to fill the template.

Guided Orchestration

While template-based orchestration presents the user with a service composition, guided orchestration allows the user to create a new service composition from scratch. Here the user chooses each service and how they connect together. With guided orchestration the queries are based on the particular input or output which must be filled at that step in the process.

We will use the landing plan composition for an example of guided orchestration. To start, the user chooses a service to start the composition. In this case, the user starts with the current velocity service by performing the query shown in Table 7.7 and selecting one of the results, in

Query:	SELECT ?service
	WHERE
	{
	?service rdf:type geo:MapService .
	?service geo:hasOutputParameter ?output .
	?output geo:hasTheme geo:AerialImagery .
	}

Result:	NAIPService CitySphereSerivce DoDImageryService

Table 7.5: Template orchestration query for a aerial imagery service.

Query:	SELECT ?service
	WHERE
	{
	?service rdf:type geo:ImageCompositionService .
	}

Result:	ImageOverlayService

Table 7.6: Template orchestration query for an image composition service.

our case the NOAACurrentVelocityService. Now the user adds services to the inputs and outputs of the NOAACurrentVelocityService. The service has two inputs, one for bathymetry and one for wind velocity. These are discovered using the queries in Table 7.8 and Table 7.9. These queries are automatically generated using the definition of the inputs for the NOAACurrentVelocityService. The inputs specify parameters with a grid encoding and bathymetry/wind velocity theme. The automatically generated queries specify only these properties to search for.

Query:	SELECT ?service
	WHERE
	{
	?service rdf:type geo:ModelService .
	?service geo:hasOutputParameter ?output .
	?output geo:hasTheme geo:CurrentVelocity .
	}

Result:	NAVOCurrentVelocityService NOAACurrentVelocityService

Table 7.7: Guided orchestration for navigation service.

Query:	SELECT ?service
	WHERE
	{
	?service geo:hasOutputParameter ?output .
	?output geo:hasTheme geo:Bathymetry .
	?output geo:hasEncoding ?output_encoding .
	?output_encoding rdf:type geo:GridEncoding .
	}

Result:	NAVOBathymetryService NOAABathymetryService

Table 7.8: Guided orchestration for bathymetry input to current model.

Query:	SELECT ?service
	WHERE
	{
	?service geo:hasOutputParameter ?output .
	?output geo:hasTheme geo:WindVelocity .
	?output geo:hasEncoding ?output_encoding .
	?output_encoding rdf:type geo:GridEncoding .
	}

Result:	NAVOWindVelocityService NWSWindVelocityService

Table 7.9: Guided orchestration for wind velocity input to current model.

Query:	SELECT ?service
	WHERE
	{
	?service geo:hasInputParameter ?input .
	?input geo:hasEncoding ?input_encoding .
	?input_encoding rdf:type geo:GridEncoding .
	?input geo:hasTheme geo:CurrentVelocity .
	?service rdf:type geo:RenderingService .
	}

Result:	CurrentArrowRendererService CurrentShadedRendererService

Table 7.10: Guided orchestration for rendering output to current model.

Next the user must add a service to process the output of the NOAACurrentVelocityService, a rendering service for this example. The user selects the output of NOAACurrentVelocityService and a query is auto-generated for all services which have an input parameter with type grid encoding and theme current velocity. The result of this query returns a number of undesired results. All the model services which take current velocity data as input are part of the results along with the desired rendering services. In order to counteract the overload of results, the guided orchestration system allows the user to modify the query at any given stage. The user adds filter conditions on the query that reduce the query results to a smaller set of desired services. In our example, the user adds the condition that the results must be of type RenderingService. This query and results are shown in Table 7.10. The CurrentArrowRendererService is chosen to draw the input current grid.

The final two services to add to the composition are the navigation service and the image composition that creates the final landing plan. The navigation chart service is discovered by a user query without any guidance as shown in Table 7.11. The composition service is discovered by building an automatic query from two service outputs, the navigation chart service and the current rendering service. Here again the user is provided with the ability to customize the query. Rather than add filters to the auto-generated query, the user removes filters from the query. Both the navigation chart service and the current rendering service have themes attached to their outputs. The image composition service does not use these data themes and such a query does not return

Query:	SELECT ?service
	WHERE
	{
	?service rdf:type geo:MapService .
	?service geo:hasOutputParameter ?output .
	?output geo:hasTheme geo:NavigationChart .
	}

Result:	RNCService DNCImageService

Table 7.11: Guided orchestration for navigation chart service.

Query:	SELECT ?service
	WHERE
	{
	?service rdf:type geo:CompositionService .
	?service geo:hasInputParameter ?input .
	?input geo:hasEncoding ?input_encoding .
	?input_encoding rdf:type geo:ImageEncoding .
	?service geo:hasInputParameter ?input2 .
	?input2 geo:hasEncoding ?input2_encoding .
	?input2_encoding rdf:type geo:ImageEncoding .
	}

Result:	ImageOverlayService

Table 7.12: Guided orchestration for image composition service with render currents and navigation chart inputs.

any results. In order to expand the query to discover the generic image composition service, the user removes the theme filters from the query. The result is shown in Table 7.12.

As shown by the example, guided orchestration requires significant input from the user to be successful. The user must have a clear picture of the overall dataflow for the composition in order to choose initial services. The user must also be able to modify the auto-generated queries in order to properly expand or contract the query result set. Lastly, the user must be able to select the correct service from the result set of each query. In the choice between a NOAA produced bathymetry service and a NAVO produced bathymetry service there may not be significant differences. But the difference between a shaded grid rendering and a vector arrow grid rendering is significant. The resulting product is quite different depending on the choice of renderer. An uninformed choice

of renderer could potentially create an erroneous product. However, the knowledge required by the user to perform guided orchestration is domain knowledge, not expert proficiency in the construction and use of Web services. For users with a large amount of domain knowledge, geospatial data and processing in our case, guided orchestration assists in the creation of compositions by removing the need for a deep understanding of Web services. For those users without enough domain knowledge to successfully use guided orchestration, template-based orchestration is the better alternative.

A fully automatic orchestration example is not presented here because such a system is not practical for a realistic architecture. Implementing a fully automatic orchestration system where correctness of the composition is guaranteed is unrealistic. If a user must correct a composition then the process becomes another form of semi-automatic orchestration. An implemented fully automatic orchestration process would execute similarly to guided orchestration. The primary difference is that a user is not available to make the choices identified in the guided orchestration example. In fully automatic orchestration the user does not modify queries to ensure useful results are returned or select the most appropriate service from a query result set. The lack of a user makes it likely that the resulting composition will be incorrect. The only way to ensure a fully automatic orchestration system creates a composition which matches a user provided goal is to specify the goal in so much detail that the orchestration system becomes a template-based system rather than a fully automatic system.

Chapter 8

Performance Analysis and Improvement

Performance of the geospatial Web service catalog and the backing model are an important consideration in determining their effectiveness for supporting automatic orchestration. Service composition algorithms place a large load on the catalog when creating orchestrations. Performance limitations within the service catalog will cascade through the composition process, making it useless.

One consequence of using an OWL ontology to encode the geospatial service model is that any compliant reasoner may be used when querying the model. Therefore, overall query performance will be dependent on the choice of reasoner. Test results presented here all use the Pellet reasoner on a 2.93 GHz Mac Pro workstation with 16GB of RAM. Unless otherwise noted, all tests were single-threaded.

8.1 Computational Complexity of Model Realization

While performance will vary between reasoning engines, the underlying computational complexity of realization does not. The computational complexity of ontology realization is determined by the class of description logic to which it belongs. Description logic classes are defined by their supported operators. Our geospatial service model is properly identified as a $\mathcal{ALCHOQ}$ description logic. Each letter in the description logic type refers to the properties available in the logic class.

The initials $\mathcal{AL}$ stand for "attributive language" and refer to the minimal class of interest [4]. The subsequent letters refer to the availability of these specific properties:

- $\mathcal{C}$: complex concept negation
- $\mathcal{H}$: role hierarchy
- $\mathcal{O}$: nominals
- $\mathcal{Q}$: qualified cardinality restrictions

Realization of an $\mathcal{ALCHOQ}$ description logic is ExpTime-Complete [38]. Generally speaking, exponential complexity is a flag that this algorithm may have performance problems.

The Pellet reasoner used in our model experiments supports both $\mathcal{SHOIN}^{(\mathcal{D})}$ (OWL-DL) and $\mathcal{SROIQ}^{(\mathcal{D})}$ (a subset of OWL2) model types. The definitions of each description logic property are defined below.

- $\mathcal{S}$: the properties of $\mathcal{ALC}$ plus role transitivity
- $\mathcal{I}$: role inverses
- $\mathcal{N}$: unqualified cardinality restrictions
- $\mathcal{R}$: complex role inclusion; reflexivity and irreflexivity; role disjointness

$\mathcal{SHOIN}^{(\mathcal{D})}$ and $\mathcal{SROIQ}^{(\mathcal{D})}$ have realization complexity of NExpTime-Complete. The lack of role inverses in the geospatial model reduces its complexity.

8.2 Core Model Realization

The design of the geospatial service model partitions the core model definition from the definitions of the service individuals. As a result, we are able to measure core model realization separately from the realization of our catalog data. Separating these two components of our ontology is important because, in practice, these pieces should not be realized together. The core model must

	μ (s)	σ (s)
Unrealized model	2.53	0.35
Realized model	1.26	0.31

Table 8.1: Realization times for the core model and the pre-realized core model.

only be realized when it is changed. Core model changes will occur much less frequently than changes to the service catalog data (individuals in our model).

Realization of the core model is necessary prior to realization of service individuals. This realization may be done at the same time as the service individuals or prior to the realization of service individuals. It is important to measure the cost to realize the core model and the cost to realize the model after it has already been realized. Timing results are shown in Table 8.1. Note that realization of a pre-realized model still has a significant cost in comparison to the original unrealized model. This cost is the realization overhead: the amount of time necessary to load the ontology and calculate that no possible new inferred axioms exist.

Given the infrequency of updates to the core model, pre-realizing the model and using that for individual realization makes the most sense. While this will add the need to synchronize model updates with service queries, it does reduce the cost of the model realization component of individual realizations by 50%.

8.3 Individual Realization

Individual realization is a more performance sensitive issue than the core model realization. Individuals are much more likely to be added or updated in our architecture, causing the cost of realizing individuals to weigh more heavily on overall system performance. As discussed earlier, the computational complexity of realization is of exponential order. Realizing a large number of service individuals may result in unacceptable performance.

	μ (s)	σ (s)
Pre-realized core model	4327	795
Unrealized core model	4426	874

Table 8.2: Performance results for realization of the full graph of service individuals. Results are shown for realization with a pre-reailzed core model and an unrealized core model.

8.3.1 Full Graph Realization

The standard method for ontology realization is to realize the entire ontology graph at once. This full realization includes both a core ontology and all individuals based on it. As discussed above, the core ontology may be pre-realized in order to improve performance of the individual realization process. Both pre-realized and unrealized core ontology tests have been performed to identify the extent of the performance difference pre-realization provides when realizing individuals. For this test a representative set of services were used. These service individuals spanned the majority of the service types available in the model, with a weighting towards the more frequently used data creation services.

The results, shown in Table 8.2, show that full graph realization is costly, regardless of whether the core model is pre-realized or not. Neither full-graph realization results provide the performance necessary for on-demand service catalog queries. All further testing uses a pre-realized core model.

8.3.2 Single Entity Realization

The high cost of realizing the entire collection of individuals at once presents a problem when creating a useful service catalog. Both live users and an automated orchestration system will require that service queries evaluate in a reasonable period of time. It is quite obvious that execution times on the order of hours is not reasonable. In general, this problem could be insurmountable. Ontology realization depends on having the entire graph of data available so that it may properly compute all inferred axioms. However, the design of our ontology causes a service individual to be disconnected from every other service individual in the ontology graph. Thus, rather than being forced to realize the entire graph of individuals at once, each individual may be realized separately.

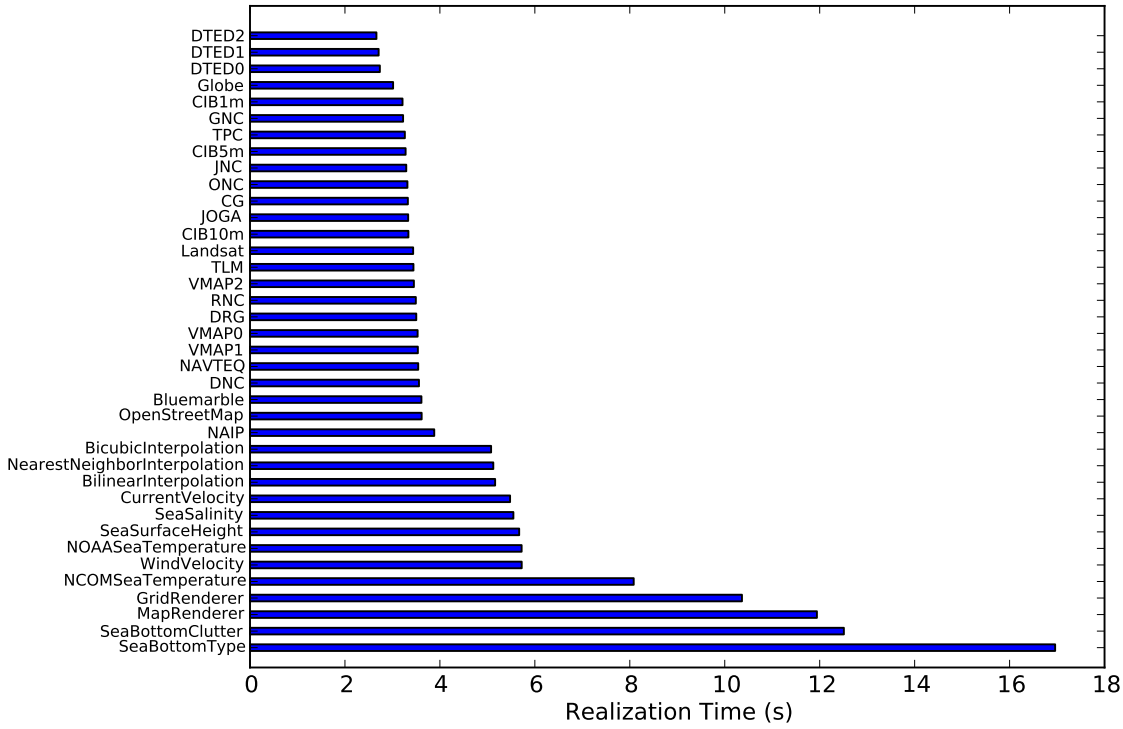


Figure 8.1: Execution times to realize each service separately.

The fully realized individuals created by the single entity realization process are identical to those created by the full graph realization.

The runtime for realization of each entity is shown in Figure 8.1. The total cost of realizing the full set of individuals is shown in Table 8.3. The cost of single entity realization is significantly lower than the cost of full graph realization. As long as individuals are not interrelated, single entity realization will be preferred over full-graph realization.

	μ (s)	σ (s)
Full graph	4327.23	795.28
Single service	186.67	5.02

Table 8.3: Performance comparison for realization of the full graph at once versus all services separately.

8.3.3 Multi-Entity Realization Performance Comparison

Based on the results shown above, it is clear that single service realization far outperforms full graph realization. However, it is possible that other graph sizes provide better performance than either full graph or single service realization. To test the effect of graph size on the realization costs, a series of realization tests was run. These tests compared performance for all service graph subsets of cardinality two. The realization times for these double service tests were compared to the costs of realizing each service separately.

The results, shown in Figures 8.2 and 8.3, shows that double service realization either outperforms or is comparable to single service realization. Complete results are located in Appendix B. The likely cause of this performance pattern is the overhead of realization. The marginal cost of adding another service to the graph for realization is much lower than the underlying base cost of performing realization. As a result, it was important to discover the point at which increases in the size of the individual graph reduced performance.

The realization costs for the double service tests vary in patterns similar to the single service tests. The most complicated services are the ones which take the longest to realize. Data creation services (map, feature, and coverage) are the least expensive to realize, while the model and processing services are the most expensive.

Running a set of exhaustive tests while the ontology individual graph size and the specific individual combinations vary would be prohibitively expensive. Instead, we measured the time to realize the entire set of individuals by partitioning the individual graph size. While the specific differences in realization time between different combinations of services would not be revealed with this series of tests, it would allow us to determine when the performance of graph realization began dropping with size. To ensure that specific combinations of services did not systematically weight the results, the tests were repeated with a randomized partitioning of the services.

Figure 8.4 shows the results from the partitioned realization tests. Each solid bar represents the mean of for a test while the error bar is its standard deviation. Partition sizes from 11 to 18 have both higher mean costs and larger standard deviations than partition sizes 10 and lower.

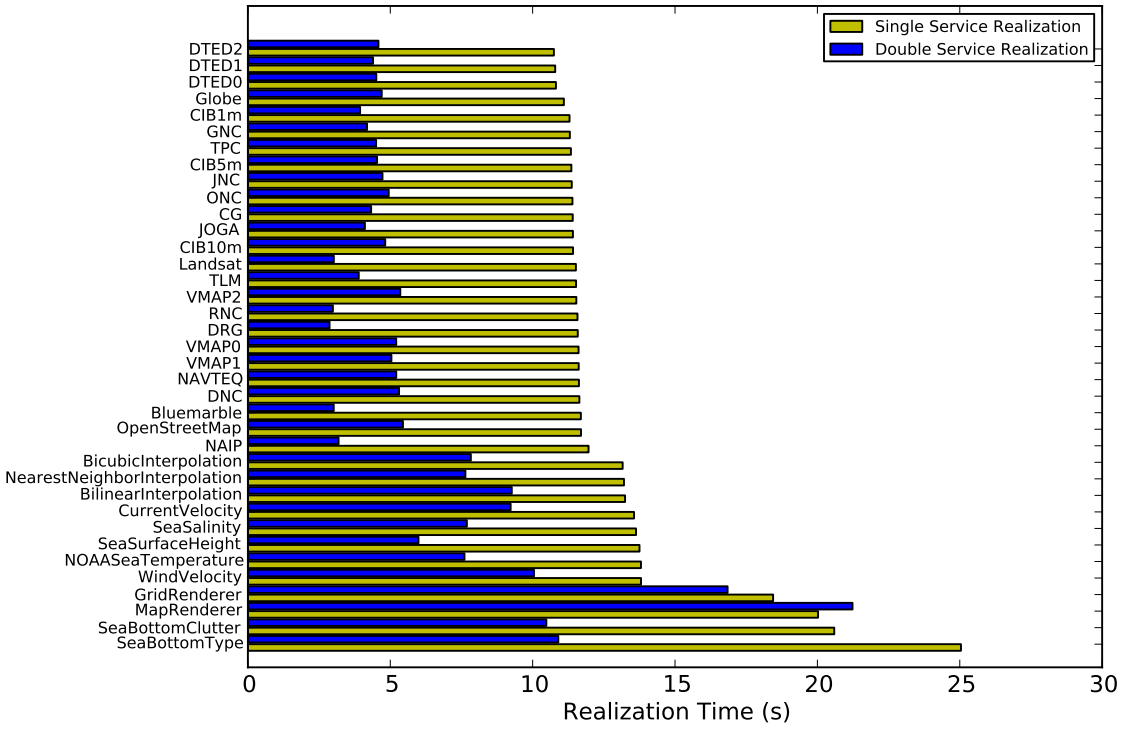


Figure 8.2: Double service realization times for NCOMSeaTemperature

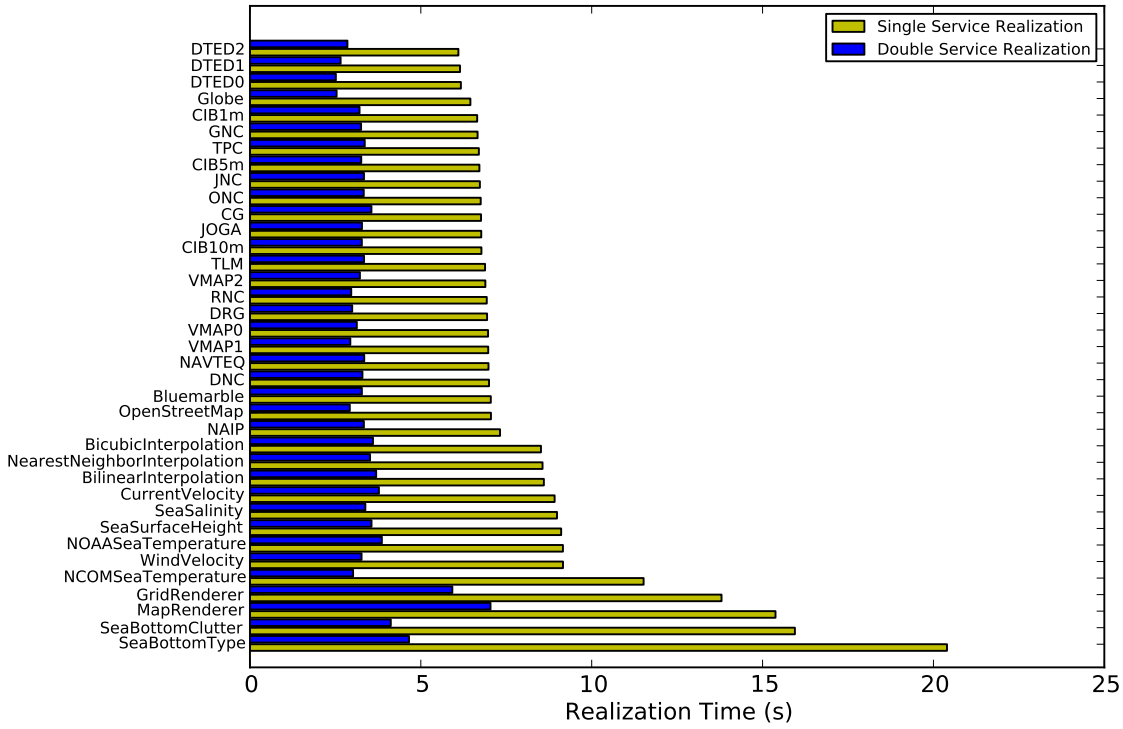


Figure 8.3: Double service realization times for Landsat

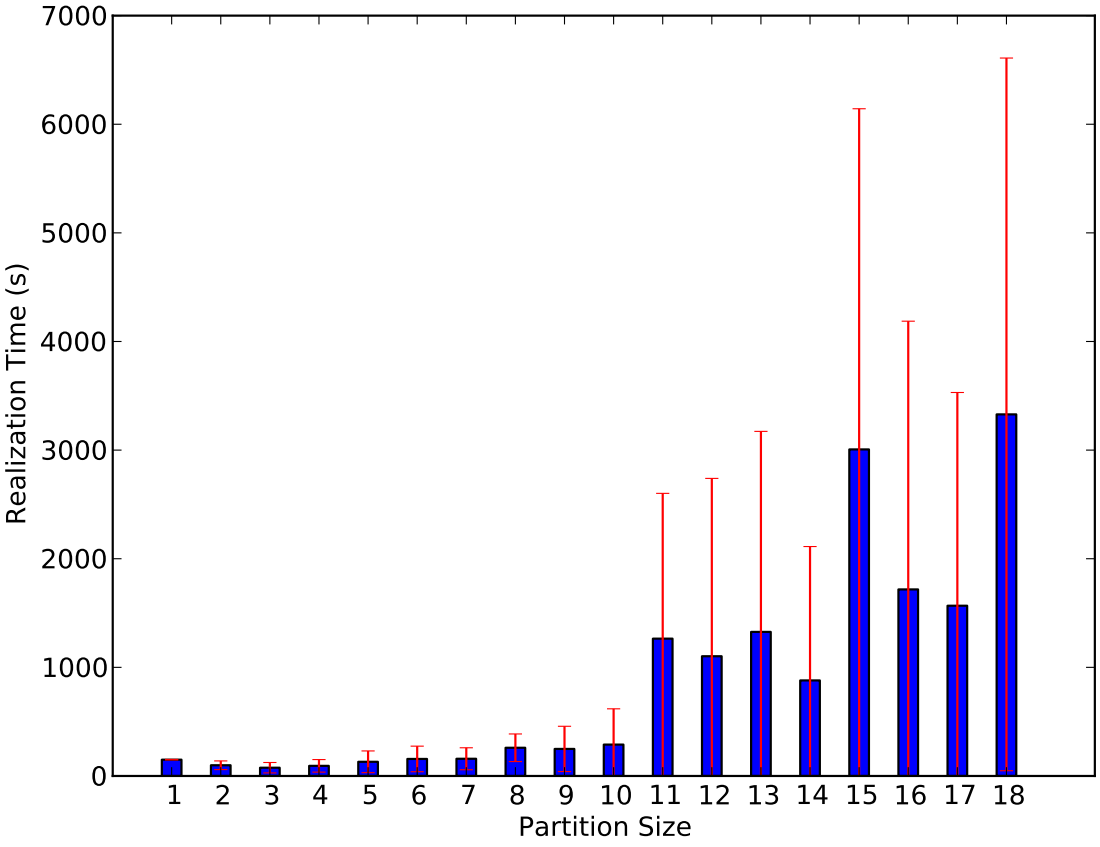


Figure 8.4: Time to realize all individuals for different sized partitions of the set of service individuals

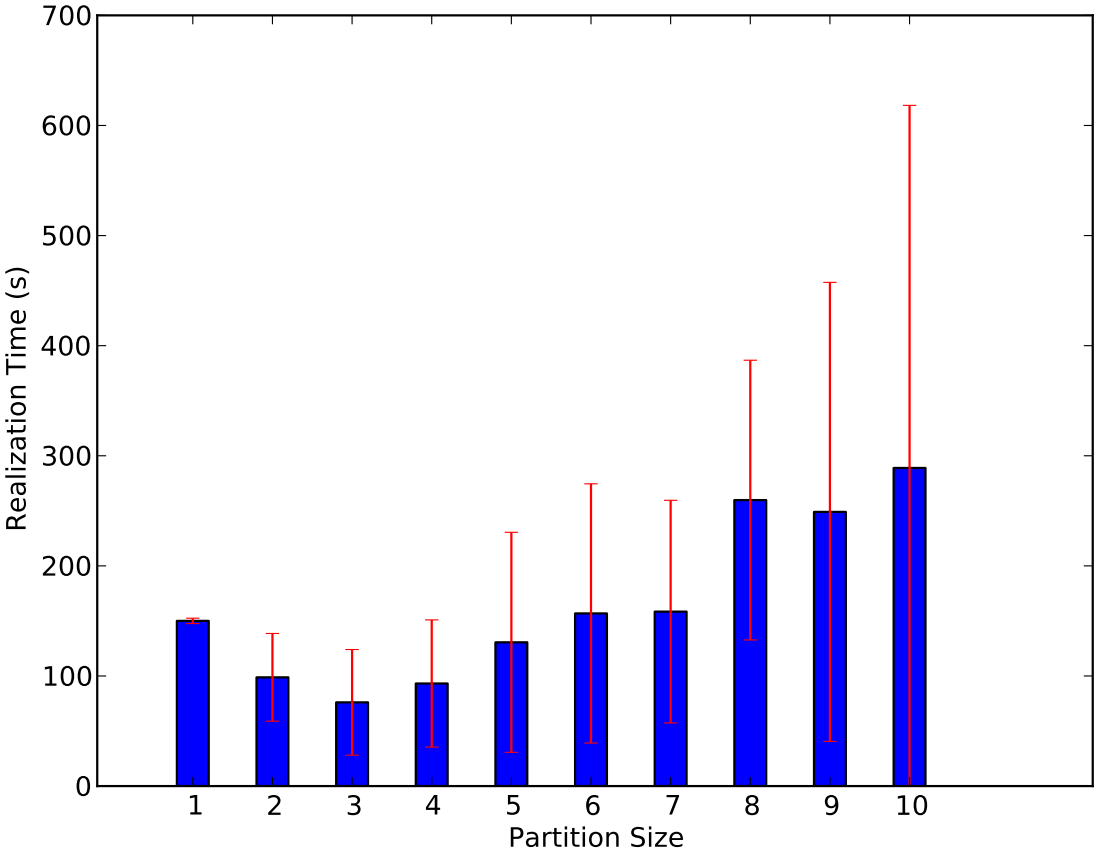


Figure 8.5: The realization times for the fastest partitions from Figure 8.4.

Figure 8.5 provides a expanded view of the results from the smaller partitions. Partition size 3 has the lowest realization time for the entire individual set, about half the cost of realizing each service individually. The results also show the standard deviations growing steadily larger as the partition size increases (also seen in Figure 8.4. The increasing standard deviation indicates that specific combinations of services increase realization time. The results from the two-service tests corroborate these results.

8.3.4 Threaded Realization

Currently available ontological reasoners do not support multi-threaded processing. This fact presents a problem for performance growth, since much of the improvements in processing recently, and projected into the future, are based on increasing the number of processing cores available to concurrently run tasks. In general, there is no fix for the lack of concurrency in ontology realization beyond radically redesigning inference algorithms. However, the unique properties of our geospatial service model allow service individuals to be realized separately, as in the partitioned realization method. Just as importantly, this property of the model also allows concurrent realization of individuals.

The method for concurrent realization is based on partitioned realization. Rather than sequentially realize each partition, they may be processed concurrently in separate threads. As expected, the results do improve as the number of concurrent threads increases.

Figure 8.6 shows the mean realization times for concurrent realization of the full data set. The results show an increase in performance for threaded realization. However, the mean performance increase in not predictable. Two threads and sixteen threads provide the best mean performance. Both four and eight threads have lower performance, though still faster than single threaded realization. Figure 8.7 shows both the median and minimum realization times for these threaded tests. The minimum time does decrease as the number of threads increases, but the only major drop is between single and double threaded execution. The median times show that eight threads actually had poorer overall performance than four threads, even though they had similar mean times. The

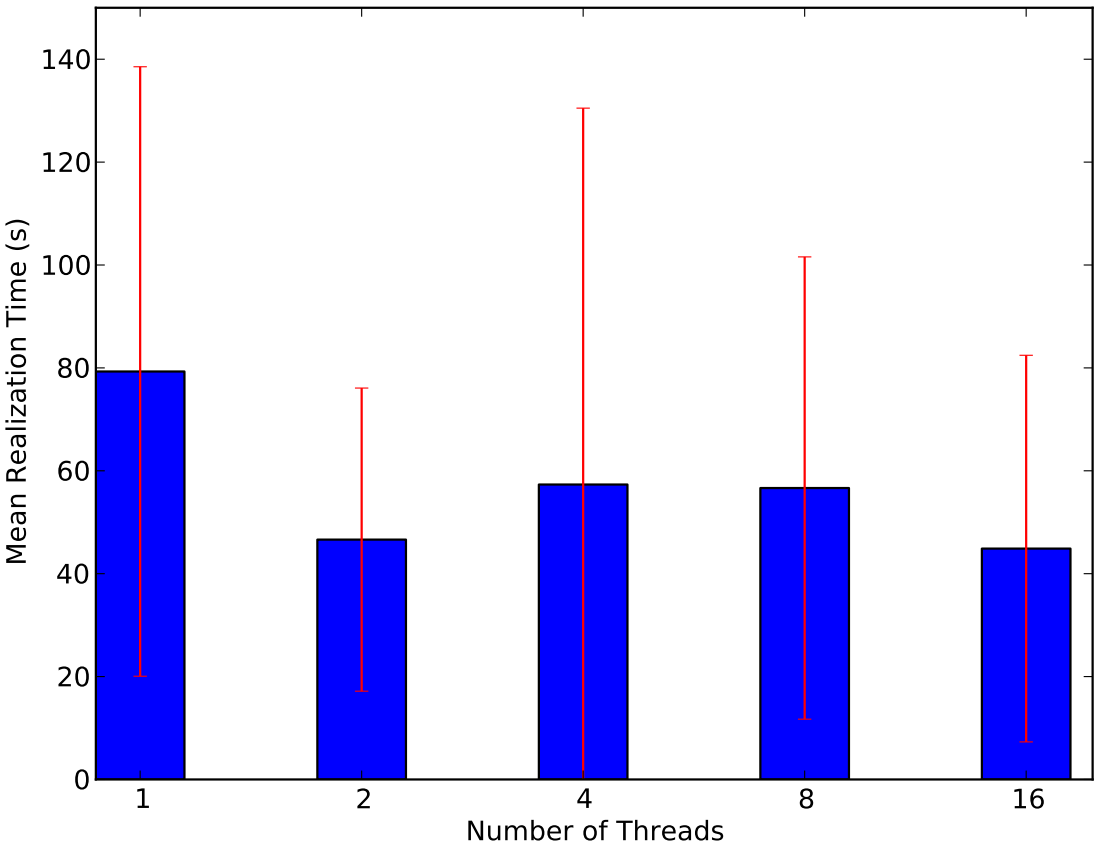


Figure 8.6: Threaded realization of cardinality 3 partitions.

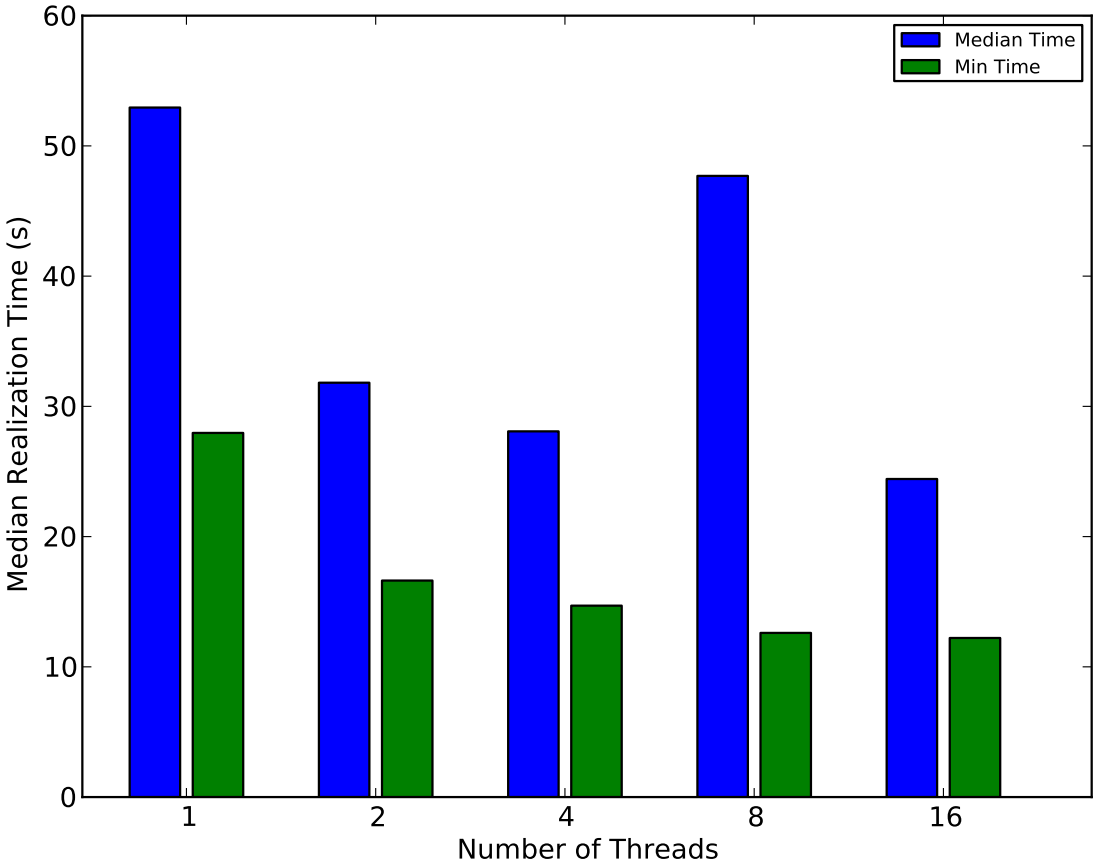


Figure 8.7: Median threaded realization of cardinality 3 partitions.

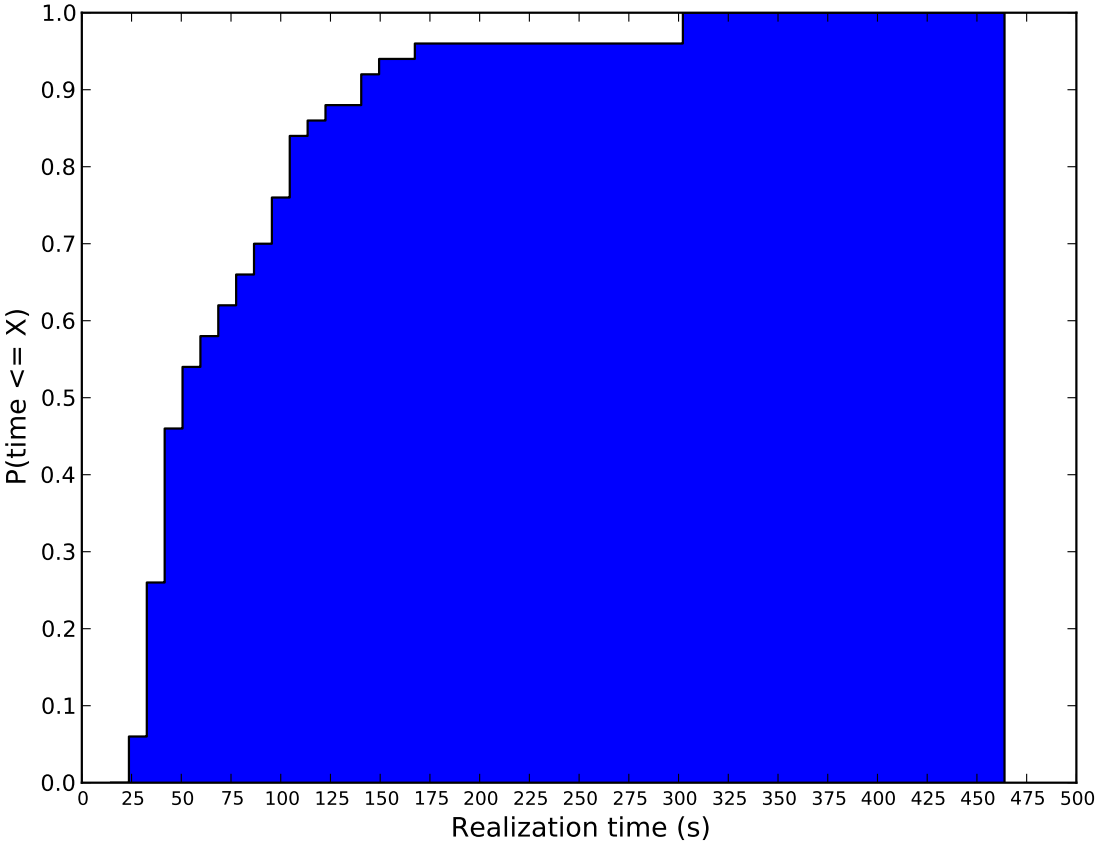


Figure 8.8: Cumulative histogram of realization times for 1 thread with 3 partitions.

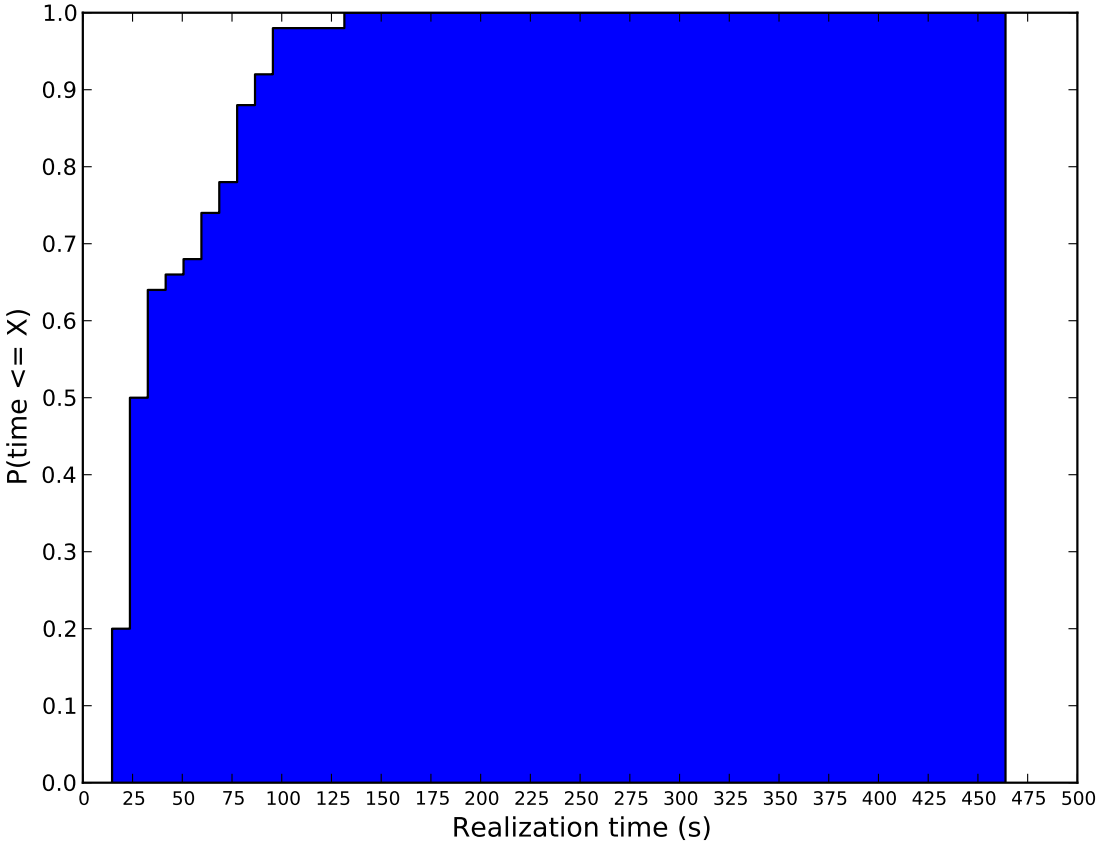


Figure 8.9: Cumulative histogram of realization times for 2 threads with 3 partitions.

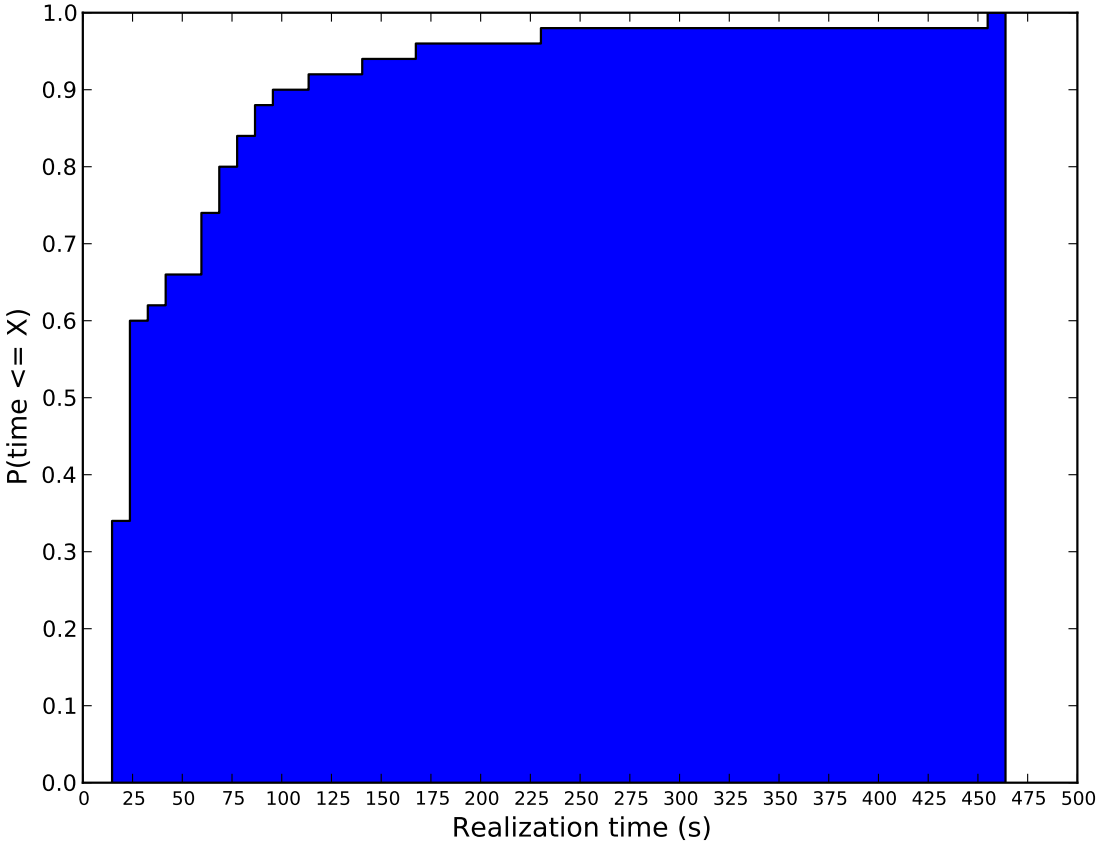


Figure 8.10: Cumulative histogram of realization times for 4 threads with 3 partitions.

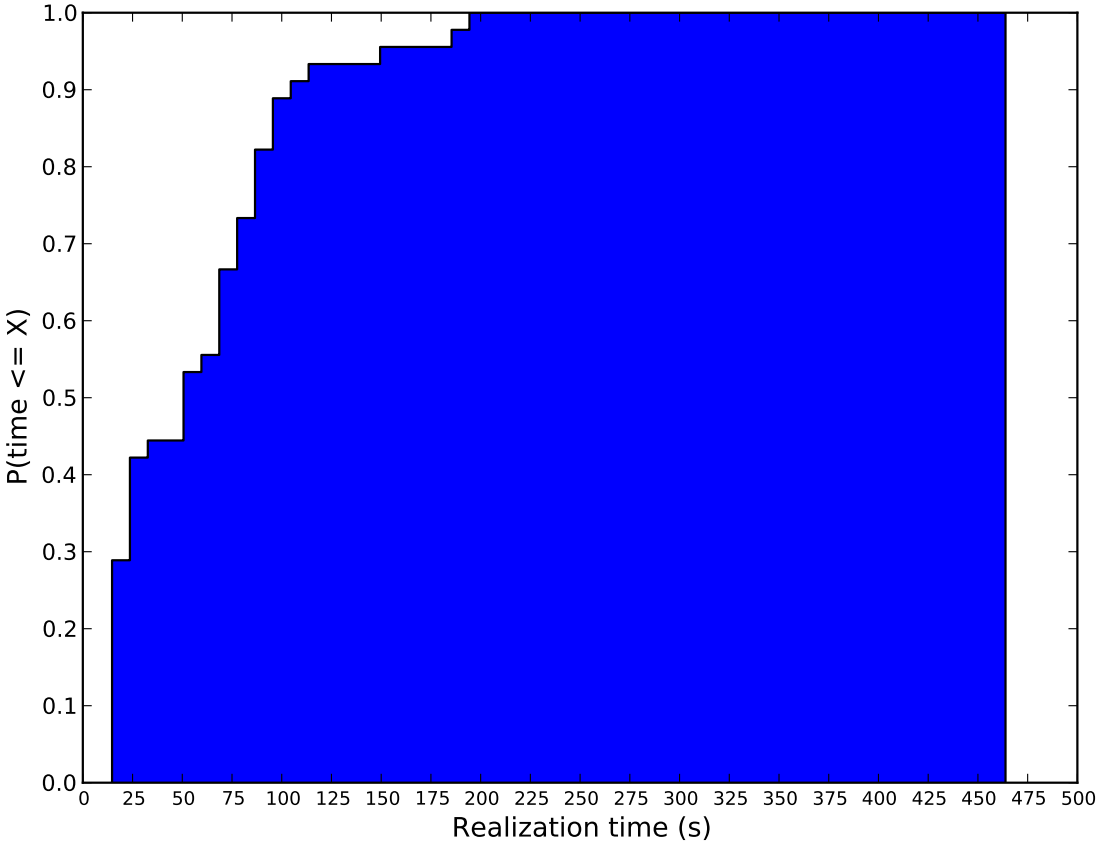


Figure 8.11: Cumulative histogram of realization times for 8 threads with 3 partitions.

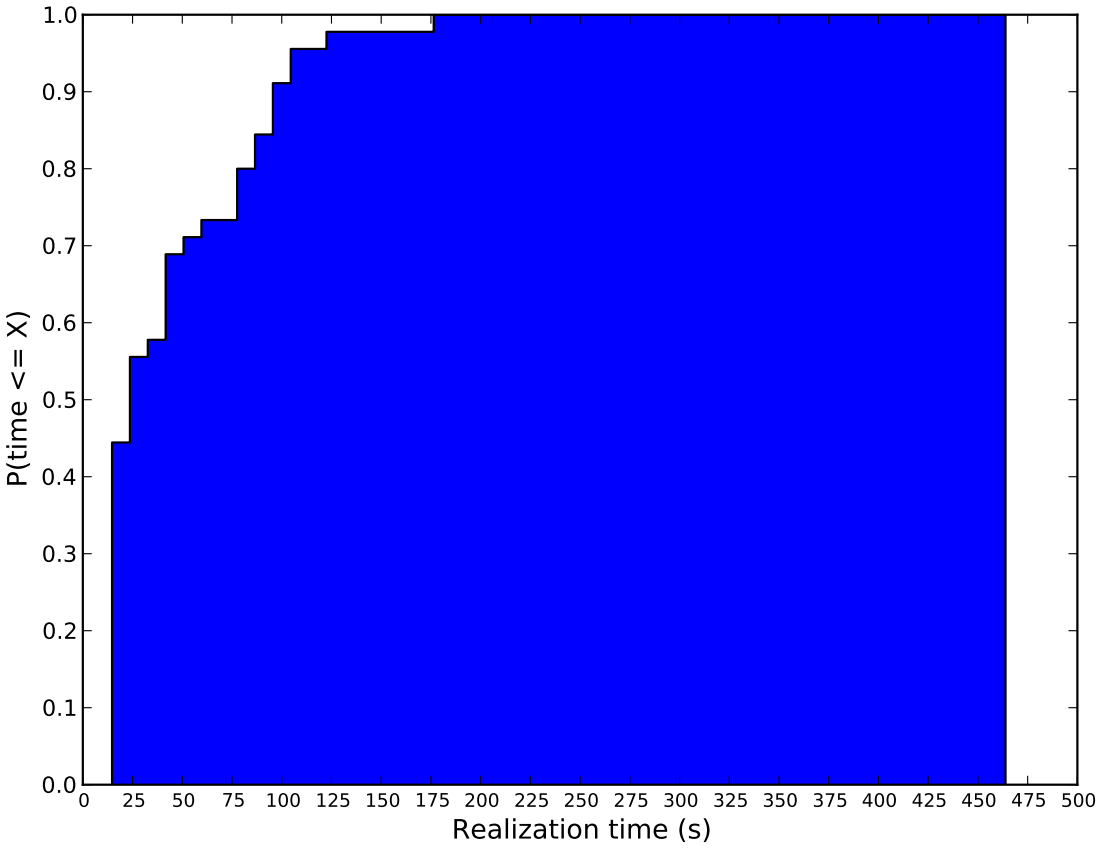


Figure 8.12: Cumulative histogram of realization times for 16 threads with 3 partitions.

four thread tests had a large outlier, where as the eight thread tests had more evenly distributed realization times which were larger than the other multi-threaded tests. Figures 8.8 - 8.12 are normalized, cumulative histograms which show similar results.

Multi-threaded realization of our data does not currently provide scaling performance improvements. The high variance in realization times combined with unpredictability of performance gains as concurrency increases limits the usefulness of multi-threading as a fix for ontology realization performance. However, these tests also show that multi-threaded data realization is useful. The multi-threaded tests all outperformed the single threaded test, both in mean, median, and minimum realization times. The histograms show a better performance distribution as well. Also important is the fact that multi-threaded realization is possible. While it does not provide a performance breakthrough, it does provide flexibility in designing the realization strategy for the geospatial service catalog, as discussed in the next section.

8.4 Realization Strategies for the Geospatial Service Catalog

A useful ontology based query system must have realization performed prior to any query evaluation. As shown earlier, the usefulness of basing query evaluation on ontology realization lies in the significant expansion in how data may be queried and evaluated, such as the ability to query over hierarchies. These additional capabilities exceed those provided by standard relational database systems. However, it is necessary to ensure that the computational costs of an ontology-based service catalog do not overwhelm the benefits of its added capabilities.

Our service catalog requires a fully realized ontology prior to query evaluation. Without a realized ontology there would be no mechanism to query based on inferred properties, removing the primary reason for using an ontology-backed catalog in the first place. There are two methods of realizing the service ontology to support query evaluation: just-in-time realization and ahead-of-time realization.

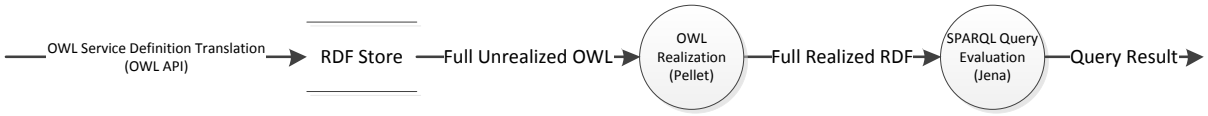


Figure 8.13: Dataflow for JIT realization.

8.4.1 Just-In-Time Realization

Just-in-time (JIT) realization performs ontology realization whenever a query is evaluated (Figure 8.13). The benefit of JIT realization is that every query is evaluated using a fully realized, up-to-date ontology. Any time the ontology is updated, either the core or the individuals, the changes are used instantly for any subsequent queries. Users benefit because they can immediately test the results of their changes, ensuring the correctness of the update. The testability of the ontology is important because users will be frequently updating the set of service individuals. Not being able to see the results of their changes could cause confusion among users and decrease their acceptance of the system.

JIT realization also simplifies the concurrency management in the service catalog. In conjunction with a properly designed atomic update process, no coordination need be done between the update and query portions of the catalog. Prior to an atomic update, query evaluation uses the older version of the ontology, and after an atomic update query evaluation automatically switches to the newer version of the ontology.

Of course, the use of JIT realization assumes that the realization process can be executed in real-time with every query. As shown in the performance sections, real-time realization is not possible. Even using our optimized partitioned evaluation strategy, the cost of realization is too high to be performed on every query. The costs are especially untenable given the large number of repeated queries necessary for any Web service orchestration process. Thus, while there are benefits to the JIT realization process, the performance issue outweighs them.

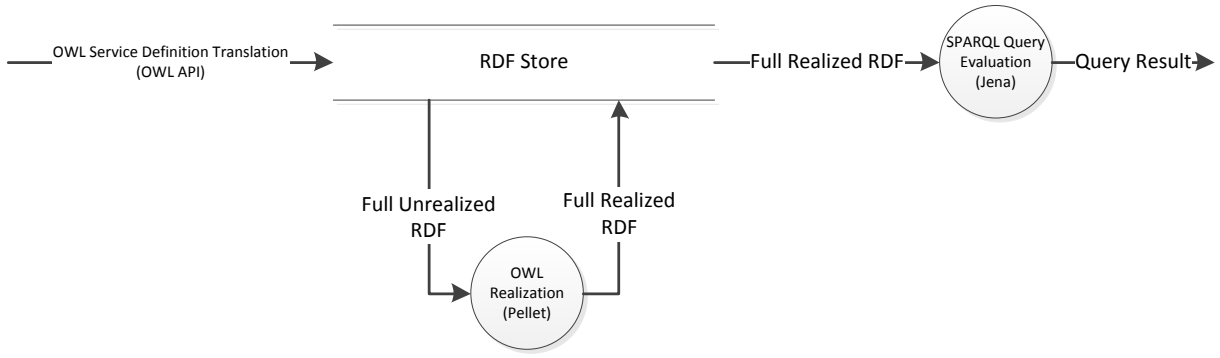


Figure 8.14: Dataflow for AOT realization.

8.4.2 Ahead-Of-Time Realization

The most obvious alternative to JIT realization is Ahead-of-Time (AOT) realization. AOT realization performs the realization process immediately after any update is made to the ontology, both to the core and to individuals (Figure 8.14). The primary reason for using AOT realization is performance. AOT realization is run in the background allowing other catalog operations, such as queries, to be evaluated without delay. Given the high cost of fully realizing the ontology, AOT realization improves responsiveness for users and orchestration systems.

Of course, AOT realization lacks the many useful features of JIT realization. Users do not have immediate access to changes they make to the ontology, which may cause problems for users as they both update and query the service catalog.

With AOT realization, the catalog must manage concurrency within the update and query processes. Whereas JIT realization simply uses a newly realized ontology for every query, an AOT realization catalog shares the realized ontology between every query process. The catalog must coordinate the use of both a pre-update realized ontology and a post-update realized ontology. Additionally, the catalog must account for occasions where the ontology is updated while a realization process is ongoing, either blocking the second realization until the first is complete or canceling the first and starting the second. In the former case, a sequence of catalog updates will queue up and suffer from long realization delays. In the latter case, a sequence of catalog updates will prevent

any updates to the realized ontology, perhaps indefinitely if a steady stream of updates continue to occur. These concurrency issues add complexity to the design of the catalog. While proper design and implementation will overcome these issues, they result in a complex and potentially error prone catalog.

8.4.3 Incremental Realization

The previous two methods of model realization assume the entire catalog is realized at once and, as we have seen, realization of all service individuals is a costly process. However, the updates to the catalog are limited to individual services rather than the whole set of services. Thus, it makes sense to only perform realization on the part of the model that has changed, rather than the full complement of service individuals. Incremental realization of full ontologies is an active area of research and not currently implemented by many reasoners. Those with incremental realization support consider it experimental. Therefore, performing incremental realization at the reasoner level is not advisable.

However, the design of our model does allow incremental realization at the catalog level. As discussed in the performance sections, service individuals are disconnected from each other in the ontology graph. In addition to allowing partitioned realization of service individuals, the disconnected nature of service individuals allows the use of incremental realization whenever the catalog is updated with a new service individual. This process is termed catalog level incremental realization because the catalog determines what has changed and should be newly realized rather than the reasoner (Figure 8.15). The catalog removes old realized ontology statements which have been updated and replace them with the newly realized ontology statements. From an implementation perspective, incremental realization is a version of AOT realization with a much faster realization process.

Incremental realization in the service catalog results in a significant performance improvement over AOT realization as can be seen by comparing the single service realization costs in Figure 8.1 to the complete realization costs shown in Figure 8.4. Separating realizations between individual

Figure 8.15: Dataflow for Incremental realization.

service updates provides the user with a constant feedback look. By blocking the catalog update process while incremental realization is being performed, the user knows exactly when the update has completed and is able to query the results immediately. The cost of realizing an individual service is small enough that blocking during updates is reasonable. In cases where there are a number of updates to apply, each may be realized and made available for querying as it is available.

There are fewer concurrency issues for incremental realization as compared to AOT realization. Updates to different service individuals do not affect each other. The incremental realization of both individuals can occur with neither affecting the other. AOT realization did not have this property. As a result, a continuous sequence of updates will not prevent completion of realization indefinitely nor cause long wait times for individual updates to be realized as with AOT realization. The multi-threaded approach was validated earlier as working correctly with our chosen reasoner. It also provided a performance benefit to the realization process, though not one that scaled linearly with the number of threads.

Incremental realization is our preferred method of managing updates in the service catalog because it provides many of the user and implementation benefits of JIT realization while not reducing the performance of catalog query evaluation. It fits well with a parallel realization approach, reduces system complexity, and provides the most interactive experience for users.

8.5 Query Performance

The previous sections deal solely with the performance of geospatial service catalog updates, but catalog queries actually impact the performance of an orchestration system the most (given the

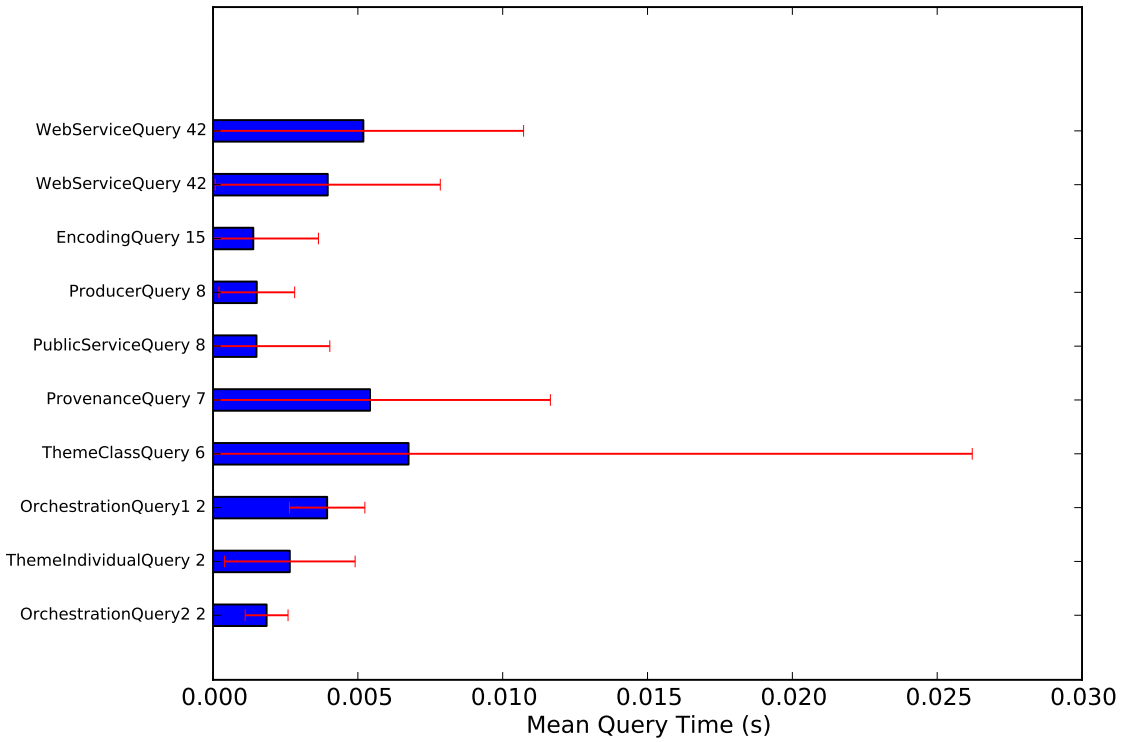


Figure 8.16: Evaluation time for queries from pre-realized RDF. Query times are shown for different queries. The number of results returned from the query are next to the query name.

untenability of JIT realization). Creating a Web service composition requires a large number of service queries. Figure 8.16 presents the execution time for a number of SPARQL catalog queries. The results show that SPARQL query performance over a pre-realized RDF graph is quite good. The mean query time for the costliest query is 8ms. Even taking into account the high standard deviation, it is highly unlikely a query would require over 40ms to execute. In that case, an orchestration with 1000 queries will only take 40s, completely reasonable for a composition of that size. The results also show that the cost is not directly related to the size of the result set. Thus a geospatial service catalog built using incremental realization will provide the necessary performance for an automatic orchestration system or user interactive query service.

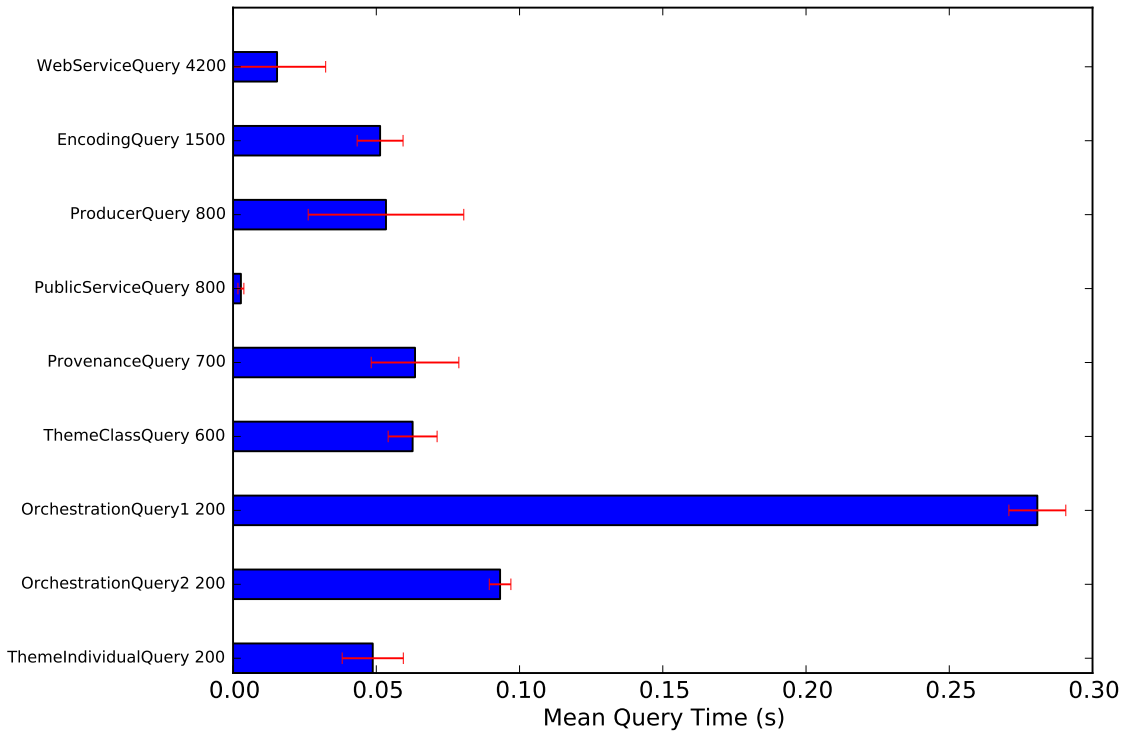


Figure 8.17: Evaluation time for queries from pre-realized RDF. Query times are shown for different queries. The number of results returned from the query are next to the query name.

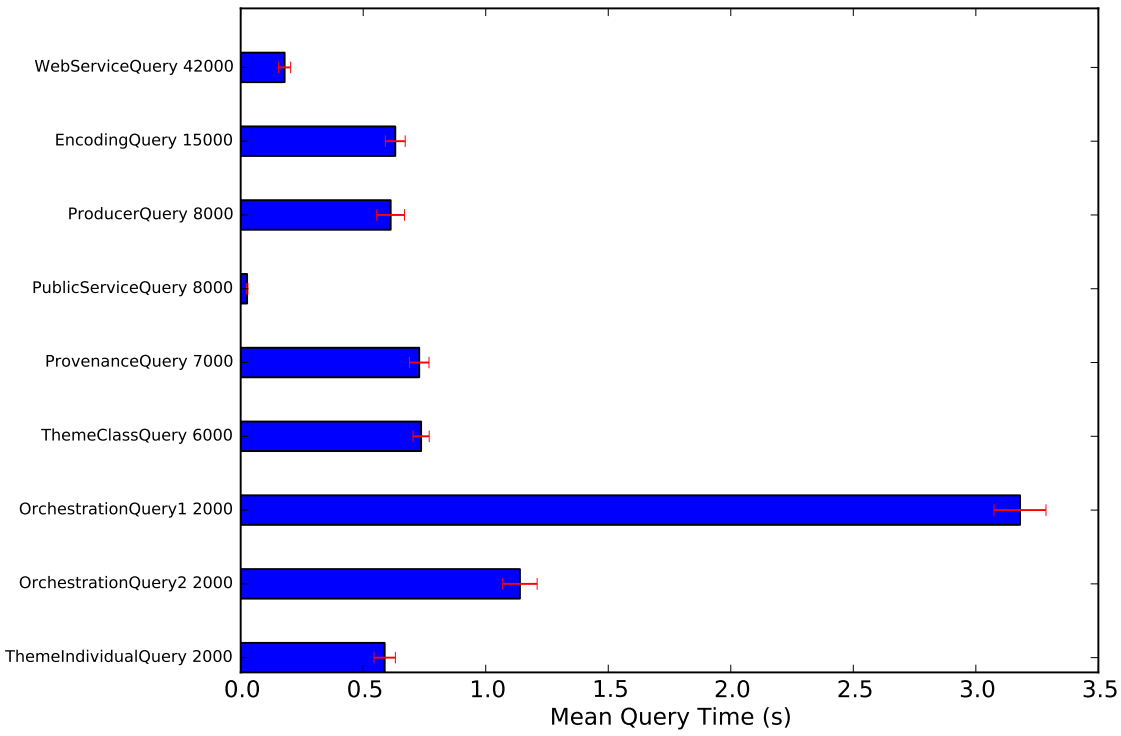


Figure 8.18: Evaluation time for queries from pre-realized RDF. Query times are shown for different queries. The number of results returned from the query are next to the query name.

Chapter 9

Conclusion

The ability to create and execute compositions of Web services has been a persistent goal since their original introduction. The major roadblock has not been the lack of tools for representing or running Web service compositions. Orchestration engines and composition languages provide true support for specifying compositions and using them as individual services. The major difficulty has always been the task of Web service discovery. Finding services that provide a requested function and have a given set of properties is non-trivial. The standard mechanism to describe a Web service, WSDL, has not changed significantly over time. The syntactic description provided by a WSDL document does not provide enough information for proper Web service discovery. Newer, and less well supported, standards such as Semantic Annotations for WSDL (SAWSDL) or OWL-S provide mechanisms to improve the way Web services are described. However, these extensions are simply mechanism which append semantics to the standard syntactic description. They do not improve the way Web services are modeled. As a result, these extensions are little used and have not improved mechanisms for Web service discovery.

Similarly, Web service catalogs have not improved greatly either. The original catalog for Web services was UDDI, which is inadequate for the purposes of Web service discovery within an orchestration system. UDDI adds no computer-readable descriptive capability over WSDL, and its attempt to model Web services amounts to a phone book taxonomy. Unfortunately, no other catalog standard has been adopted which improves upon it.

The lack of an effective model for Web services is at the heart of these problems. Creating a useful model for all possible Web services is currently not practicable. It is unrealistic to expect that anything more than a simple taxonomic model of all Web services is possible.

The approach taken in this work is a far better alternative: create a domain model for geospatial Web services rather than all Web services. By limiting the focus of the model to the geospatial service domain, we were able to add much more functionality than in a generic service model. The unique properties of geospatial services are properly accounted for in the model and catalog. These properties, such as the stateless nature of geospatial services or the complex data types they use, heavily influenced the design of the geospatial service model and catalog.

The semantic modeling approach taken in this work differs from the standard approaches for semantic annotation of Web services. The model created here was designed with the specific goal of supporting key functions of a geospatial Web service catalog. These functions include supporting thematic queries, data input/output parameter queries, data type and encoding queries, and service definition validation. By approaching the service modeling task with these requirements we created a model with far more functionality than is provided by a basic thematic ontology vocabulary for services. To meet the catalog requirements, the geospatial service model provides automatic classification, service definition validation, and parameter annotations in addition to a service theme taxonomy. The resulting model facilitates reuse of services in the architecture, supports resource sharing with other architectures, and allows logical validation of both the model itself and service definitions.

By motivating the model design with geospatial service catalog requirements, we reduced the complexity of the design and implementation of the geospatial Web service catalog. By using the existing automated inference capabilities of descriptive logics (OWL in our case), the catalog is able to both validate and automate service definition updates as well as evaluate complex hierarchical queries not supported by traditional relational query designs. The performance of existing inference and SPARQL query engines meets the needs of a service catalog intended for implementation in user service discovery and automatic orchestration systems. As improvements are made

to these reasoning and query engines, they can easily be incorporated into the geospatial service catalog. Similarly, improvements to the geospatial service model are easily incorporated into the catalog and integrate with the existing reasoning and query components. The resulting catalog provides an improved capability for querying services and surpasses the ability of existing catalog systems to support orchestration systems.

There are a number of areas for further research and development of the geospatial Web service model. The geospatial model presented here provides a framework usable in our current Web service architecture. However, expanding it to support a wider array of individuals is necessary for use in other architectures. The model itself can also be improved for performance. Currently, the model is a $\mathcal{ALCHOQ}$ description logic. Redesigning it as a different, less complex type of model may be possible. If so, it may improve the computational complexity of realization. Additionally, research is necessary on tuning the model for existing realization algorithms. These model improvements would not necessarily change computational complexity, but target the capabilities of the current state-of-the-art in realization algorithms.

Further possible research on the geospatial Web service catalog includes integrating the design created here with the OGC Catalog Service for the Web (CSW) standard. The OGC CSW is a flexible standard for a geospatial service catalog which may be able to support the geospatial model and query functionality defined in this work. The CSW standard supports integrating a variety of data models and schemas for representing data and services. There have been initial attempts at adding a semantic capability to the CSW. Adding the semantic model designed here would significantly improve the ability of CSW to model and query geospatial services. At the same time, the CSW standard would provide a mechanism to integrate our work into existing geospatial architectures.

Appendix A

Geospatial Service Ontology

```
<?xml version="1.0"?>


<!DOCTYPE Ontology [
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY xml "http://www.w3.org/XML/1998/namespace" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>


<Ontology xmlns="http://www.w3.org/2002/07/owl#"
     xml:base="uri://geocomp.nrlssc.navy.mil/geoservice.owl"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:xml="http://www.w3.org/XML/1998/namespace"
     ontologyIRI="uri://geocomp.nrlssc.navy.mil/geoservice.owl">
    <Prefix name="rdf" IRI="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
    <Prefix name="rdfs" IRI="http://www.w3.org/2000/01/rdf-schema#"/>
    <Prefix name="xsd" IRI="http://www.w3.org/2001/XMLSchema#"/>
    <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
    <Declaration>
        <Class abbreviatedIRI="owl:Thing"/>
    </Declaration>
    <Declaration>
        <Class IRI="#CADRGService"/>
    </Declaration>
    <Declaration>
        <Class IRI="#CIBService"/>
    </Declaration>
    <Declaration>
        <Class IRI="#Chart"/>
    </Declaration>
    <Declaration>
        <Class IRI="#Civilian"/>
    </Declaration>
    <Declaration>
        <Class IRI="#Commercial"/>
    </Declaration>
    <Declaration>
        <Class IRI="#Community"/>
    </Declaration>
    <Declaration>
        <Class IRI="#CoverageEncoding"/>
    </Declaration>
```

```
<Declaration>
    <Class IRI="#CoverageParameter"/>
</Declaration>
<Declaration>
    <Class IRI="#CoverageService"/>
</Declaration>
<Declaration>
    <Class IRI="#DataOrigin"/>
</Declaration>
<Declaration>
    <Class IRI="#DataProducer"/>
</Declaration>
<Declaration>
    <Class IRI="#DataTheme"/>
</Declaration>
<Declaration>
    <Class IRI="#Elevation"/>
</Declaration>
<Declaration>
    <Class IRI="#FeatureDetectionService"/>
</Declaration>
<Declaration>
    <Class IRI="#FeatureExtractionService"/>
</Declaration>
<Declaration>
    <Class IRI="#FeatureService"/>
</Declaration>
<Declaration>
    <Class IRI="#GeospatialEncoding"/>
</Declaration>
<Declaration>
    <Class IRI="#GeospatialParameter"/>
</Declaration>
<Declaration>
    <Class IRI="#GeospatialService"/>
</Declaration>
<Declaration>
    <Class IRI="#GridEncoding"/>
</Declaration>
<Declaration>
    <Class IRI="#GridParameter"/>
</Declaration>
<Declaration>
    <Class IRI="#ImageCompositionService"/>
</Declaration>
<Declaration>
    <Class IRI="#ImageEncoding"/>
</Declaration>
<Declaration>
    <Class IRI="#ImageParameter"/>
</Declaration>
<Declaration>
    <Class IRI="#ImageProcessingService"/>
</Declaration>
<Declaration>
    <Class IRI="#ImageRenderingService"/>
</Declaration>
<Declaration>
    <Class IRI="#Imagery"/>
</Declaration>
<Declaration>
    <Class IRI="#Interpolated"/>
</Declaration>
<Declaration>
    <Class IRI="#InterpolationService"/>
</Declaration>
<Declaration>
    <Class IRI="#METOC"/>
```

```
</Declaration>
<Declaration>
    <Class IRI="#MapService"/>
</Declaration>
<Declaration>
    <Class IRI="#Measured"/>
</Declaration>
<Declaration>
    <Class IRI="#Military"/>
</Declaration>
<Declaration>
    <Class IRI="#Model"/>
</Declaration>
<Declaration>
    <Class IRI="#ModelService"/>
</Declaration>
<Declaration>
    <Class IRI="#NCOMService"/>
</Declaration>
<Declaration>
    <Class IRI="#PointVectorParameter"/>
</Declaration>
<Declaration>
    <Class IRI="#PolygonVectorParameter"/>
</Declaration>
<Declaration>
    <Class IRI="#PolylineVectorParameter"/>
</Declaration>
<Declaration>
    <Class IRI="#Processed"/>
</Declaration>
<Declaration>
    <Class IRI="#ProcessedImage"/>
</Declaration>
<Declaration>
    <Class IRI="#ProcessingService"/>
</Declaration>
<Declaration>
    <Class IRI="#PublicService"/>
</Declaration>
<Declaration>
    <Class IRI="#Reprojected"/>
</Declaration>
<Declaration>
    <Class IRI="#SeaBottom"/>
</Declaration>
<Declaration>
    <Class IRI="#ServiceParameter"/>
</Declaration>
<Declaration>
    <Class IRI="#SizeDecreased"/>
</Declaration>
<Declaration>
    <Class IRI="#SizeIncreased"/>
</Declaration>
<Declaration>
    <Class IRI="#USGovernment"/>
</Declaration>
<Declaration>
    <Class IRI="#UserParameter"/>
</Declaration>
<Declaration>
    <Class IRI="#VectorEncoding"/>
</Declaration>
<Declaration>
    <Class IRI="#VectorParameter"/>
</Declaration>
<Declaration>
```

```
    <Class IRI="#WebService"/>
</Declaration>
<Declaration>
    <ObjectProperty abbreviatedIRI="owl:topObjectProperty"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasEncoding"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasInputParameter"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasOrigin"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasOutputParameter"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasProducer"/>
</Declaration>
<Declaration>
    <ObjectProperty IRI="#hasTheme"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#AerialImagery"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#AirTemperature"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#Airforce"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#BMP"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#Bathymetry"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#BicubicInterpolated"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#BilinearInterpolated"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#CurrentVelocity"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#DigitalGlobe"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#EarthSat"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#Forecast"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#GIF"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#GML"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#GeoEye"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#GeoTIFF"/>
</Declaration>
```

```
<Declaration>
    <NamedIndividual IRI="#HDF"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#Hindcast"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#JPEG"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#JPEG2000"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#JPG"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#KML"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#LandElevation"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#NASA"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#NAVO"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#NAVTEQ"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#NGA"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#NNInterpolated"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#NOAA"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#NRL"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#NauticalChart"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#NavigationChart"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#NetCDF"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#Nowcast"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#OSM"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#OtherNavy"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#PNG"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#PoliticalMap"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#Raw"/>
```

```
</Declaration>
<Declaration>
    <NamedIndividual IRI="#RoadMap"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#SatelliteImagery"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#SeaBottomClutter"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#SeaBottomType"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#SeaSalinity"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#SeaSurfaceHeight"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#SeaTemperature"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#Shapefile"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#TIFF"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#TopographicChart"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#USDA"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#USGS"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#UnprocessedMeasuredData"/>
</Declaration>
<Declaration>
    <NamedIndividual IRI="#WindVelocity"/>
</Declaration>
<EquivalentClasses>
    <Class IRI="#CoverageParameter"/>
    <ObjectSomeValuesFrom>
        <ObjectProperty IRI="#hasEncoding"/>
        <Class IRI="#CoverageEncoding"/>
    </ObjectSomeValuesFrom>
</EquivalentClasses>
<EquivalentClasses>
    <Class IRI="#CoverageService"/>
    <ObjectIntersectionOf>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasOutputParameter"/>
            <Class IRI="#CoverageParameter"/>
        </ObjectAllValuesFrom>
        <ObjectExactCardinality cardinality="0">
            <ObjectProperty IRI="#hasInputParameter"/>
            <Class IRI="#GeospatialParameter"/>
        </ObjectExactCardinality>
    </ObjectIntersectionOf>
</EquivalentClasses>
<EquivalentClasses>
    <Class IRI="#FeatureDetectionService"/>
    <ObjectIntersectionOf>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasInputParameter"/>
```

```
            <ObjectUnionOf>
                <Class IRI="#CoverageParameter"/>
                <Class IRI="#UserParameter"/>
            </ObjectUnionOf>
        </ObjectAllValuesFrom>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasOutputParameter"/>
            <Class IRI="#VectorParameter"/>
        </ObjectAllValuesFrom>
        <ObjectExactCardinality cardinality="1">
            <ObjectProperty IRI="#hasInputParameter"/>
            <Class IRI="#CoverageParameter"/>
        </ObjectExactCardinality>
    </ObjectIntersectionOf>
</EquivalentClasses>
<EquivalentClasses>
    <Class IRI="#FeatureExtractionService"/>
    <ObjectIntersectionOf>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasInputParameter"/>
            <ObjectUnionOf>
                <Class IRI="#ImageParameter"/>
                <Class IRI="#UserParameter"/>
            </ObjectUnionOf>
        </ObjectAllValuesFrom>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasOutputParameter"/>
            <Class IRI="#VectorParameter"/>
        </ObjectAllValuesFrom>
        <ObjectExactCardinality cardinality="1">
            <ObjectProperty IRI="#hasInputParameter"/>
            <Class IRI="#ImageParameter"/>
        </ObjectExactCardinality>
    </ObjectIntersectionOf>
</EquivalentClasses>
<EquivalentClasses>
    <Class IRI="#FeatureService"/>
    <ObjectIntersectionOf>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasOutputParameter"/>
            <Class IRI="#VectorParameter"/>
        </ObjectAllValuesFrom>
        <ObjectExactCardinality cardinality="0">
            <ObjectProperty IRI="#hasInputParameter"/>
            <Class IRI="#GeospatialParameter"/>
        </ObjectExactCardinality>
    </ObjectIntersectionOf>
</EquivalentClasses>
<EquivalentClasses>
    <Class IRI="#GeospatialService"/>
    <ObjectIntersectionOf>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasOutputParameter"/>
            <Class IRI="#GeospatialParameter"/>
        </ObjectAllValuesFrom>
        <ObjectExactCardinality cardinality="1">
            <ObjectProperty IRI="#hasOutputParameter"/>
        </ObjectExactCardinality>
    </ObjectIntersectionOf>
</EquivalentClasses>
<EquivalentClasses>
    <Class IRI="#GridParameter"/>
    <ObjectSomeValuesFrom>
        <ObjectProperty IRI="#hasEncoding"/>
        <Class IRI="#GridEncoding"/>
    </ObjectSomeValuesFrom>
</EquivalentClasses>
<EquivalentClasses>
```

```
    <Class IRI="#ImageCompositionService"/>
    <ObjectIntersectionOf>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasInputParameter"/>
            <ObjectUnionOf>
                <Class IRI="#ImageParameter"/>
                <Class IRI="#UserParameter"/>
            </ObjectUnionOf>
        </ObjectAllValuesFrom>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasOutputParameter"/>
            <Class IRI="#ImageParameter"/>
        </ObjectAllValuesFrom>
        <ObjectMinCardinality cardinality="2">
            <ObjectProperty IRI="#hasInputParameter"/>
            <Class IRI="#ImageParameter"/>
        </ObjectMinCardinality>
    </ObjectIntersectionOf>
</EquivalentClasses>
<EquivalentClasses>
    <Class IRI="#ImageParameter"/>
    <ObjectSomeValuesFrom>
        <ObjectProperty IRI="#hasEncoding"/>
        <Class IRI="#ImageEncoding"/>
    </ObjectSomeValuesFrom>
</EquivalentClasses>
<EquivalentClasses>
    <Class IRI="#ImageProcessingService"/>
    <ObjectIntersectionOf>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasInputParameter"/>
            <ObjectUnionOf>
                <Class IRI="#ImageParameter"/>
                <Class IRI="#UserParameter"/>
            </ObjectUnionOf>
        </ObjectAllValuesFrom>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasOutputParameter"/>
            <Class IRI="#ImageParameter"/>
        </ObjectAllValuesFrom>
        <ObjectExactCardinality cardinality="1">
            <ObjectProperty IRI="#hasInputParameter"/>
            <Class IRI="#ImageParameter"/>
        </ObjectExactCardinality>
    </ObjectIntersectionOf>
</EquivalentClasses>
<EquivalentClasses>
    <Class IRI="#ImageRenderingService"/>
    <ObjectIntersectionOf>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasInputParameter"/>
            <ObjectUnionOf>
                <Class IRI="#CoverageParameter"/>
                <Class IRI="#UserParameter"/>
                <Class IRI="#VectorParameter"/>
            </ObjectUnionOf>
        </ObjectAllValuesFrom>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasOutputParameter"/>
            <Class IRI="#ImageParameter"/>
        </ObjectAllValuesFrom>
        <ObjectMinCardinality cardinality="1">
            <ObjectProperty IRI="#hasInputParameter"/>
            <ObjectUnionOf>
                <Class IRI="#CoverageParameter"/>
                <Class IRI="#VectorParameter"/>
            </ObjectUnionOf>
        </ObjectMinCardinality>
```

```
    </ObjectIntersectionOf>
</EquivalentClasses>
<EquivalentClasses>
    <Class IRI="#InterpolationService"/>
    <ObjectIntersectionOf>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasInputParameter"/>
            <ObjectUnionOf>
                <Class IRI="#CoverageParameter"/>
                <Class IRI="#PointVectorParameter"/>
                <Class IRI="#UserParameter"/>
            </ObjectUnionOf>
        </ObjectAllValuesFrom>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasOutputParameter"/>
            <ObjectIntersectionOf>
                <Class IRI="#GridParameter"/>
                <ObjectAllValuesFrom>
                    <ObjectProperty IRI="#hasOrigin"/>
                    <Class IRI="#Interpolated"/>
                </ObjectAllValuesFrom>
            </ObjectIntersectionOf>
        </ObjectAllValuesFrom>
        <ObjectExactCardinality cardinality="1">
            <ObjectProperty IRI="#hasInputParameter"/>
            <ObjectUnionOf>
                <Class IRI="#CoverageParameter"/>
                <Class IRI="#PointVectorParameter"/>
            </ObjectUnionOf>
        </ObjectExactCardinality>
    </ObjectIntersectionOf>
</EquivalentClasses>
<EquivalentClasses>
    <Class IRI="#MapService"/>
    <ObjectIntersectionOf>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasOutputParameter"/>
            <Class IRI="#ImageParameter"/>
        </ObjectAllValuesFrom>
        <ObjectExactCardinality cardinality="1">
            <ObjectProperty IRI="#hasOutputParameter"/>
            <Class IRI="#ImageParameter"/>
        </ObjectExactCardinality>
        <ObjectMaxCardinality cardinality="0">
            <ObjectProperty IRI="#hasInputParameter"/>
            <Class IRI="#GeospatialParameter"/>
        </ObjectMaxCardinality>
    </ObjectIntersectionOf>
</EquivalentClasses>
<EquivalentClasses>
    <Class IRI="#ModelService"/>
    <ObjectIntersectionOf>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasInputParameter"/>
            <ObjectUnionOf>
                <Class IRI="#CoverageParameter"/>
                <Class IRI="#UserParameter"/>
                <Class IRI="#VectorParameter"/>
            </ObjectUnionOf>
        </ObjectAllValuesFrom>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasOutputParameter"/>
            <ObjectIntersectionOf>
                <Class IRI="#GridParameter"/>
                <ObjectSomeValuesFrom>
                    <ObjectProperty IRI="#hasOrigin"/>
                    <Class IRI="#Model"/>
                </ObjectSomeValuesFrom>
```

```
            </ObjectIntersectionOf>
        </ObjectAllValuesFrom>
        <ObjectMinCardinality cardinality="1">
            <ObjectProperty IRI="#hasInputParameter"/>
            <ObjectUnionOf>
                <Class IRI="#CoverageParameter"/>
                <Class IRI="#VectorParameter"/>
            </ObjectUnionOf>
        </ObjectMinCardinality>
    </ObjectIntersectionOf>
</EquivalentClasses>
<EquivalentClasses>
    <Class IRI="#ProcessingService"/>
    <ObjectSomeValuesFrom>
        <ObjectProperty IRI="#hasInputParameter"/>
        <Class IRI="#GeospatialParameter"/>
    </ObjectSomeValuesFrom>
</EquivalentClasses>
<EquivalentClasses>
    <Class IRI="#PublicService"/>
    <ObjectIntersectionOf>
        <Class IRI="#WebService"/>
        <ObjectComplementOf>
            <Class IRI="#ProcessingService"/>
        </ObjectComplementOf>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasOutputParameter"/>
            <ObjectAllValuesFrom>
                <ObjectProperty IRI="#hasProducer"/>
                <ObjectUnionOf>
                    <Class IRI="#Civilian"/>
                    <Class IRI="#Community"/>
                </ObjectUnionOf>
            </ObjectAllValuesFrom>
        </ObjectAllValuesFrom>
    </ObjectIntersectionOf>
</EquivalentClasses>
<EquivalentClasses>
    <Class IRI="#VectorParameter"/>
    <ObjectSomeValuesFrom>
        <ObjectProperty IRI="#hasEncoding"/>
        <Class IRI="#VectorEncoding"/>
    </ObjectSomeValuesFrom>
</EquivalentClasses>
<EquivalentClasses>
    <Class IRI="#WebService"/>
    <ObjectIntersectionOf>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasInputParameter"/>
            <Class IRI="#ServiceParameter"/>
        </ObjectAllValuesFrom>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasOutputParameter"/>
            <Class IRI="#ServiceParameter"/>
        </ObjectAllValuesFrom>
    </ObjectIntersectionOf>
</EquivalentClasses>
<SubClassOf>
    <Class IRI="#CADRGService"/>
    <Class IRI="#MapService"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#CIBService"/>
    <Class IRI="#MapService"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Chart"/>
    <Class IRI="#DataTheme"/>
```

```
</SubClassOf>
<SubClassOf>
    <Class IRI="#Civilian"/>
    <Class IRI="#USGovernment"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Commercial"/>
    <Class IRI="#DataProducer"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Community"/>
    <Class IRI="#DataProducer"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#CoverageEncoding"/>
    <Class IRI="#GeospatialEncoding"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#CoverageParameter"/>
    <Class IRI="#GeospatialParameter"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#CoverageService"/>
    <Class IRI="#GeospatialService"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Elevation"/>
    <Class IRI="#DataTheme"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#FeatureDetectionService"/>
    <Class IRI="#GeospatialService"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#FeatureExtractionService"/>
    <Class IRI="#GeospatialService"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#FeatureService"/>
    <Class IRI="#GeospatialService"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#GeospatialParameter"/>
    <Class IRI="#ServiceParameter"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#GeospatialService"/>
    <Class IRI="#WebService"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#GeospatialService"/>
    <ObjectAllValuesFrom>
        <ObjectProperty IRI="#hasOutputParameter"/>
        <ObjectExactCardinality cardinality="1">
            <ObjectProperty IRI="#hasProducer"/>
        </ObjectExactCardinality>
    </ObjectAllValuesFrom>
</SubClassOf>
<SubClassOf>
    <Class IRI="#GridEncoding"/>
    <Class IRI="#CoverageEncoding"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#GridParameter"/>
    <Class IRI="#CoverageParameter"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#ImageCompositionService"/>
```

```
    <Class IRI="#GeospatialService"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#ImageEncoding"/>
    <Class IRI="#GeospatialEncoding"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#ImageParameter"/>
    <Class IRI="#GeospatialParameter"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#ImageProcessingService"/>
    <Class IRI="#GeospatialService"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#ImageRenderingService"/>
    <Class IRI="#GeospatialService"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Imagery"/>
    <Class IRI="#DataTheme"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Interpolated"/>
    <Class IRI="#Processed"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#InterpolationService"/>
    <Class IRI="#GeospatialService"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#InterpolationService"/>
    <ObjectAllValuesFrom>
        <ObjectProperty IRI="#hasOutputParameter"/>
        <ObjectAllValuesFrom>
            <ObjectProperty IRI="#hasOrigin"/>
            <Class IRI="#Interpolated"/>
        </ObjectAllValuesFrom>
    </ObjectAllValuesFrom>
</SubClassOf>
<SubClassOf>
    <Class IRI="#METOC"/>
    <Class IRI="#DataTheme"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#MapService"/>
    <Class IRI="#GeospatialService"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Measured"/>
    <Class IRI="#DataOrigin"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Military"/>
    <Class IRI="#USGovernment"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Model"/>
    <Class IRI="#DataOrigin"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#ModelService"/>
    <Class IRI="#GeospatialService"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#NCOMService"/>
    <Class IRI="#ModelService"/>
</SubClassOf>
```

```
<SubClassOf>
    <Class IRI="#NCOMService"/>
    <ObjectAllValuesFrom>
        <ObjectProperty IRI="#hasOutputParameter"/>
        <ObjectIntersectionOf>
            <ObjectAllValuesFrom>
                <ObjectProperty IRI="#hasTheme"/>
                <Class IRI="#METOC"/>
            </ObjectAllValuesFrom>
            <ObjectHasValue>
                <ObjectProperty IRI="#hasProducer"/>
                <NamedIndividual IRI="#NAVO"/>
            </ObjectHasValue>
        </ObjectIntersectionOf>
    </ObjectAllValuesFrom>
</SubClassOf>
<SubClassOf>
    <Class IRI="#PointVectorParameter"/>
    <Class IRI="#VectorParameter"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#PolygonVectorParameter"/>
    <Class IRI="#VectorParameter"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#PolylineVectorParameter"/>
    <Class IRI="#VectorParameter"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Processed"/>
    <Class IRI="#DataOrigin"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#ProcessedImage"/>
    <Class IRI="#Processed"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#ProcessingService"/>
    <Class IRI="#WebService"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#PublicService"/>
    <Class IRI="#WebService"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#Reprojected"/>
    <Class IRI="#ProcessedImage"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#SeaBottom"/>
    <Class IRI="#DataTheme"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#SizeDecreased"/>
    <Class IRI="#ProcessedImage"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#SizeIncreased"/>
    <Class IRI="#ProcessedImage"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#USGovernment"/>
    <Class IRI="#DataProducer"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#UserParameter"/>
    <Class IRI="#ServiceParameter"/>
</SubClassOf>
```

```
<SubClassOf>
    <Class IRI="#VectorEncoding"/>
    <Class IRI="#GeospatialEncoding"/>
</SubClassOf>
<SubClassOf>
    <Class IRI="#VectorParameter"/>
    <Class IRI="#GeospatialParameter"/>
</SubClassOf>
<DisjointClasses>
    <Class IRI="#Chart"/>
    <Class IRI="#Elevation"/>
    <Class IRI="#Imagery"/>
    <Class IRI="#METOC"/>
    <Class IRI="#SeaBottom"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#Civilian"/>
    <Class IRI="#Military"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#Commercial"/>
    <Class IRI="#USGovernment"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#CoverageEncoding"/>
    <Class IRI="#ImageEncoding"/>
    <Class IRI="#VectorEncoding"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#CoverageParameter"/>
    <Class IRI="#ImageParameter"/>
    <Class IRI="#VectorParameter"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#DataOrigin"/>
    <Class IRI="#DataProducer"/>
    <Class IRI="#DataTheme"/>
    <Class IRI="#GeospatialEncoding"/>
    <Class IRI="#ServiceParameter"/>
    <Class IRI="#WebService"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#GeospatialParameter"/>
    <Class IRI="#UserParameter"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#Measured"/>
    <Class IRI="#Model"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#PointVectorParameter"/>
    <Class IRI="#PolygonVectorParameter"/>
    <Class IRI="#PolylineVectorParameter"/>
</DisjointClasses>
<DisjointClasses>
    <Class IRI="#ProcessingService"/>
    <Class IRI="#PublicService"/>
</DisjointClasses>
<ClassAssertion>
    <Class IRI="#Imagery"/>
    <NamedIndividual IRI="#AerialImagery"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#METOC"/>
    <NamedIndividual IRI="#AirTemperature"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Military"/>
```

```
    <NamedIndividual IRI="#Airforce"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#ImageEncoding"/>
    <NamedIndividual IRI="#BMP"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#SeaBottom"/>
    <NamedIndividual IRI="#Bathymetry"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Interpolated"/>
    <NamedIndividual IRI="#BicubicInterpolated"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Interpolated"/>
    <NamedIndividual IRI="#BilinearInterpolated"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#METOC"/>
    <NamedIndividual IRI="#CurrentVelocity"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Commercial"/>
    <NamedIndividual IRI="#DigitalGlobe"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Commercial"/>
    <NamedIndividual IRI="#EarthSat"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Model"/>
    <NamedIndividual IRI="#Forecast"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#ImageEncoding"/>
    <NamedIndividual IRI="#GIF"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#VectorEncoding"/>
    <NamedIndividual IRI="#GML"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Commercial"/>
    <NamedIndividual IRI="#GeoEye"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#GridEncoding"/>
    <NamedIndividual IRI="#GeoTIFF"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#GridEncoding"/>
    <NamedIndividual IRI="#HDF"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Model"/>
    <NamedIndividual IRI="#Hindcast"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#ImageEncoding"/>
    <NamedIndividual IRI="#JPEG"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#ImageEncoding"/>
    <NamedIndividual IRI="#JPEG2000"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#ImageEncoding"/>
```

```
    <NamedIndividual IRI="#JPG"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#VectorEncoding"/>
    <NamedIndividual IRI="#KML"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Elevation"/>
    <NamedIndividual IRI="#LandElevation"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Civilian"/>
    <NamedIndividual IRI="#NASA"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Military"/>
    <NamedIndividual IRI="#NAVO"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Commercial"/>
    <NamedIndividual IRI="#NAVTEQ"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Military"/>
    <NamedIndividual IRI="#NGA"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Interpolated"/>
    <NamedIndividual IRI="#NNInterpolated"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Civilian"/>
    <NamedIndividual IRI="#NOAA"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Military"/>
    <NamedIndividual IRI="#NRL"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Chart"/>
    <NamedIndividual IRI="#NauticalChart"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Chart"/>
    <NamedIndividual IRI="#NavigationChart"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#GridEncoding"/>
    <NamedIndividual IRI="#NetCDF"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Model"/>
    <NamedIndividual IRI="#Nowcast"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Community"/>
    <NamedIndividual IRI="#OSM"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Military"/>
    <NamedIndividual IRI="#OtherNavy"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#ImageEncoding"/>
    <NamedIndividual IRI="#PNG"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Chart"/>
```

```
    <NamedIndividual IRI="#PoliticalMap"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#CoverageEncoding"/>
    <NamedIndividual IRI="#Raw"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Chart"/>
    <NamedIndividual IRI="#RoadMap"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Imagery"/>
    <NamedIndividual IRI="#SatelliteImagery"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#SeaBottom"/>
    <NamedIndividual IRI="#SeaBottomClutter"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#SeaBottom"/>
    <NamedIndividual IRI="#SeaBottomType"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#METOC"/>
    <NamedIndividual IRI="#SeaSalinity"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#METOC"/>
    <NamedIndividual IRI="#SeaSurfaceHeight"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#METOC"/>
    <NamedIndividual IRI="#SeaTemperature"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#VectorEncoding"/>
    <NamedIndividual IRI="#Shapefile"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#ImageEncoding"/>
    <NamedIndividual IRI="#TIFF"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Chart"/>
    <NamedIndividual IRI="#TopographicChart"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Civilian"/>
    <NamedIndividual IRI="#USDA"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Civilian"/>
    <NamedIndividual IRI="#USGS"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#Measured"/>
    <NamedIndividual IRI="#UnprocessedMeasuredData"/>
</ClassAssertion>
<ClassAssertion>
    <Class IRI="#METOC"/>
    <NamedIndividual IRI="#WindVelocity"/>
</ClassAssertion>
<SameIndividual>
    <NamedIndividual IRI="#JPEG"/>
    <NamedIndividual IRI="#JPG"/>
</SameIndividual>
<SubObjectPropertyOf>
    <ObjectProperty IRI="#hasEncoding"/>
```

```
        <ObjectProperty abbreviatedIRI="owl:topObjectProperty"/>
    </SubObjectPropertyOf>
    <SubObjectPropertyOf>
        <ObjectProperty IRI="#hasOutputParameter"/>
        <ObjectProperty abbreviatedIRI="owl:topObjectProperty"/>
    </SubObjectPropertyOf>
    <FunctionalObjectProperty>
        <ObjectProperty IRI="#hasOrigin"/>
    </FunctionalObjectProperty>
</Ontology>



<!-- Generated by the OWL API (version 3.1.0.20069) http://owlapi.sourceforge.net -->
```

Appendix B

Double Service Ontology Realization Results

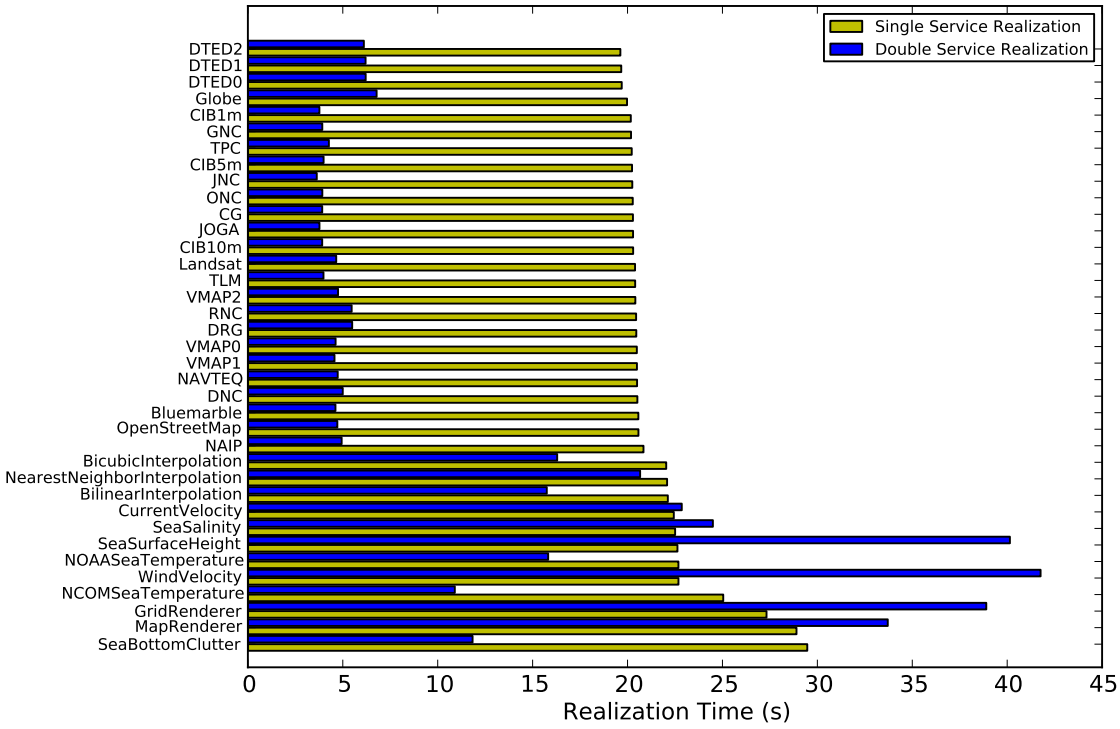


Figure B.1: Double service realization times for SeaBottomType

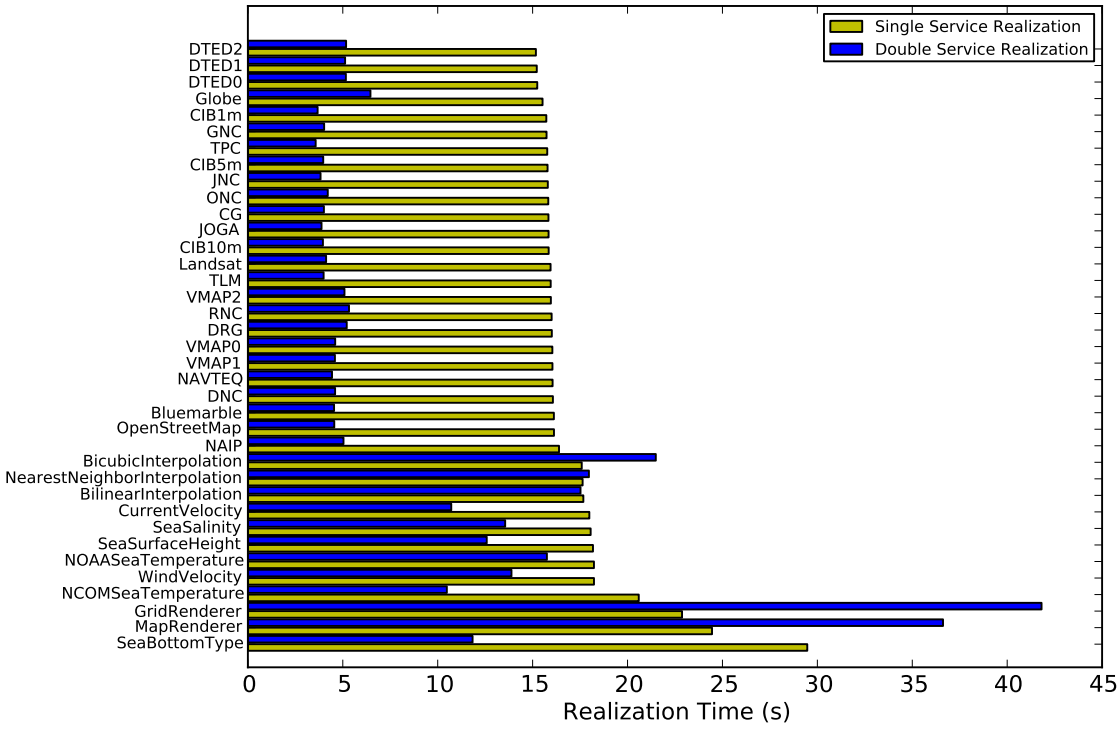


Figure B.2: Double service realization times for SeaBottomClutter

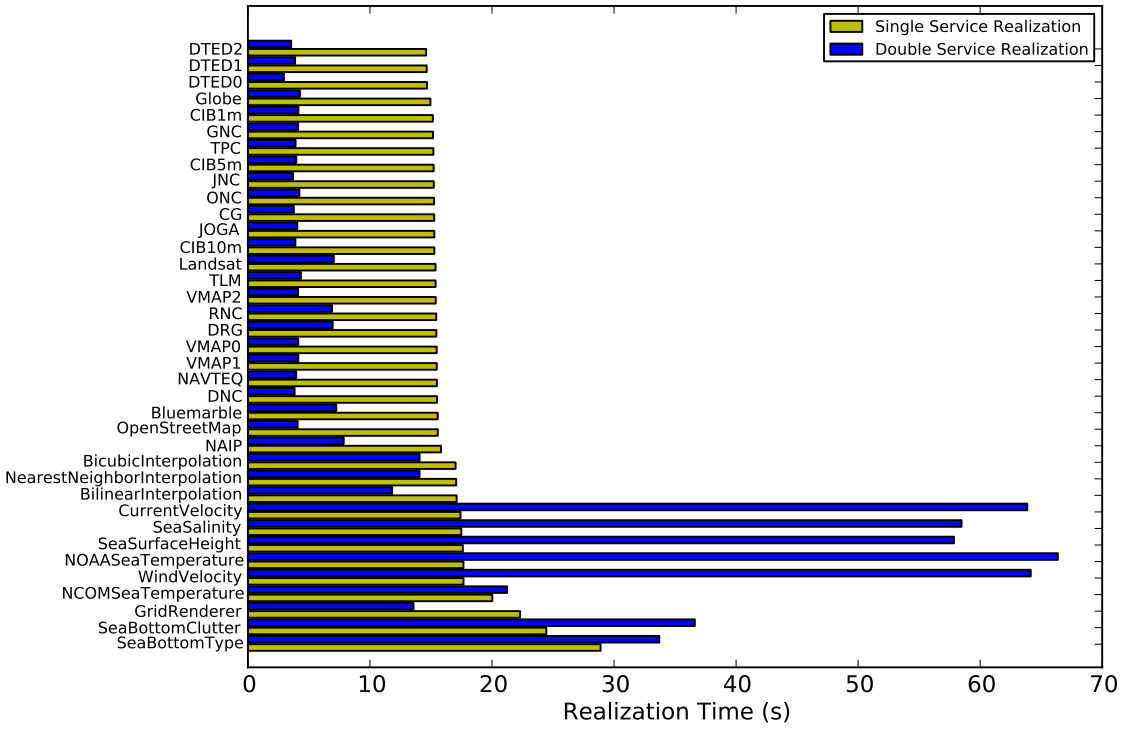


Figure B.3: Double service realization times for MapRenderer

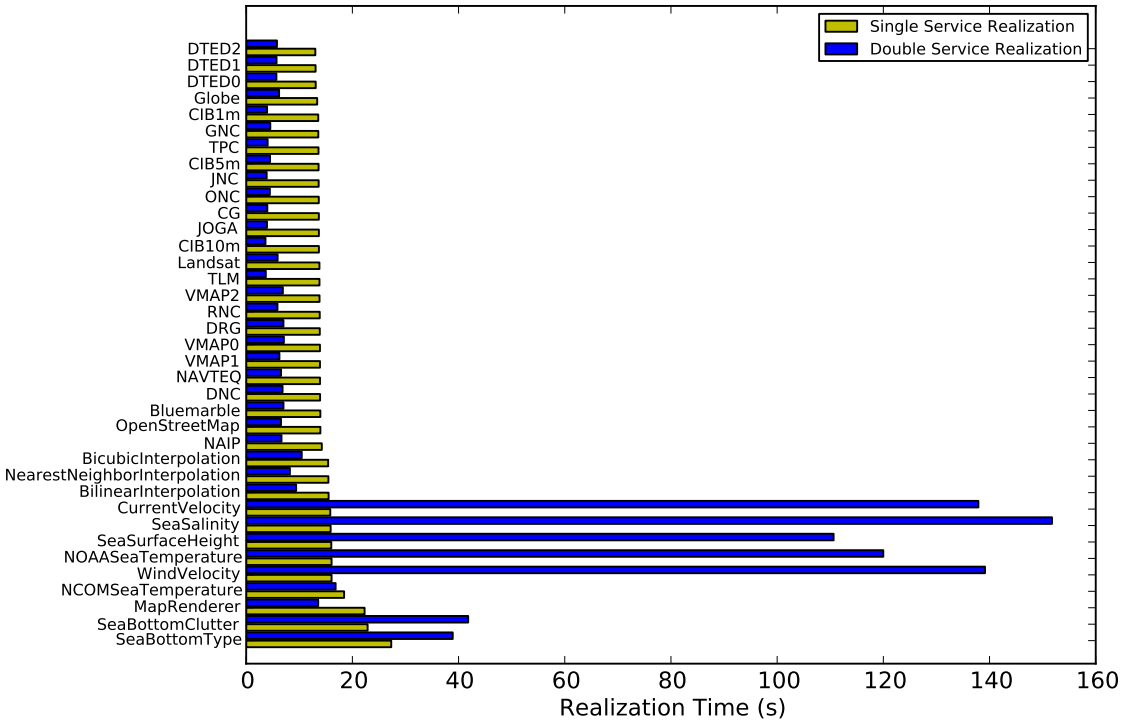


Figure B.4: Double service realization times for GridRenderer

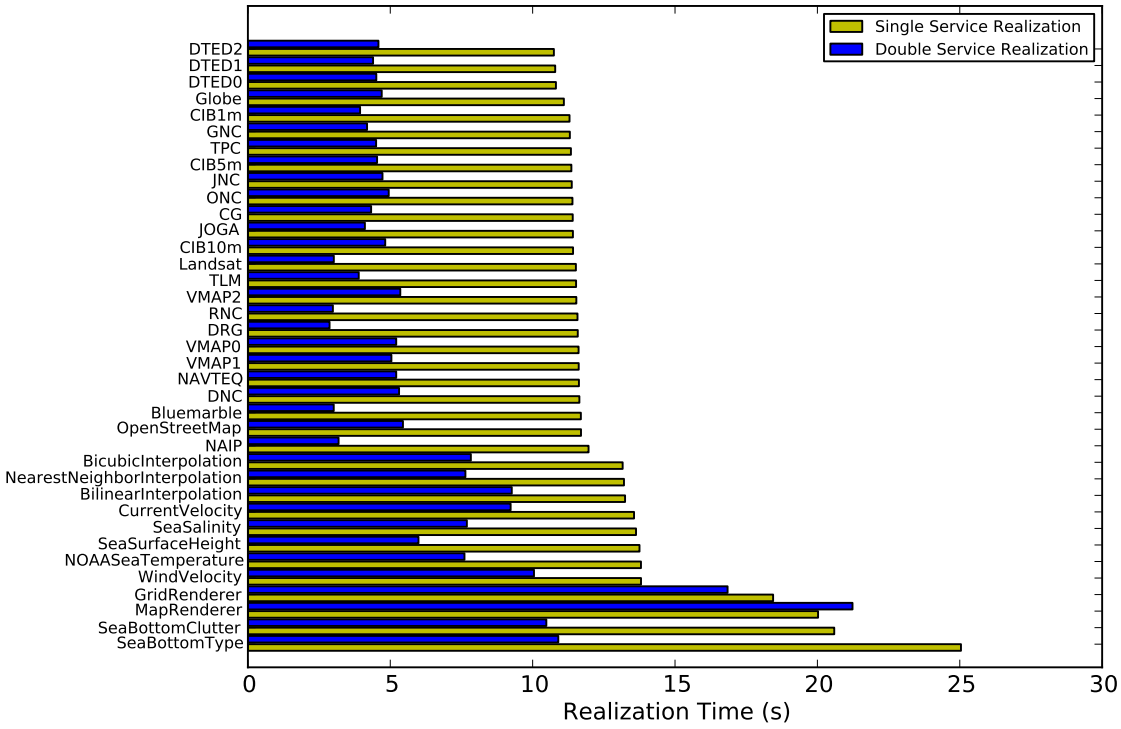


Figure B.5: Double service realization times for NCOMSeaTemperature

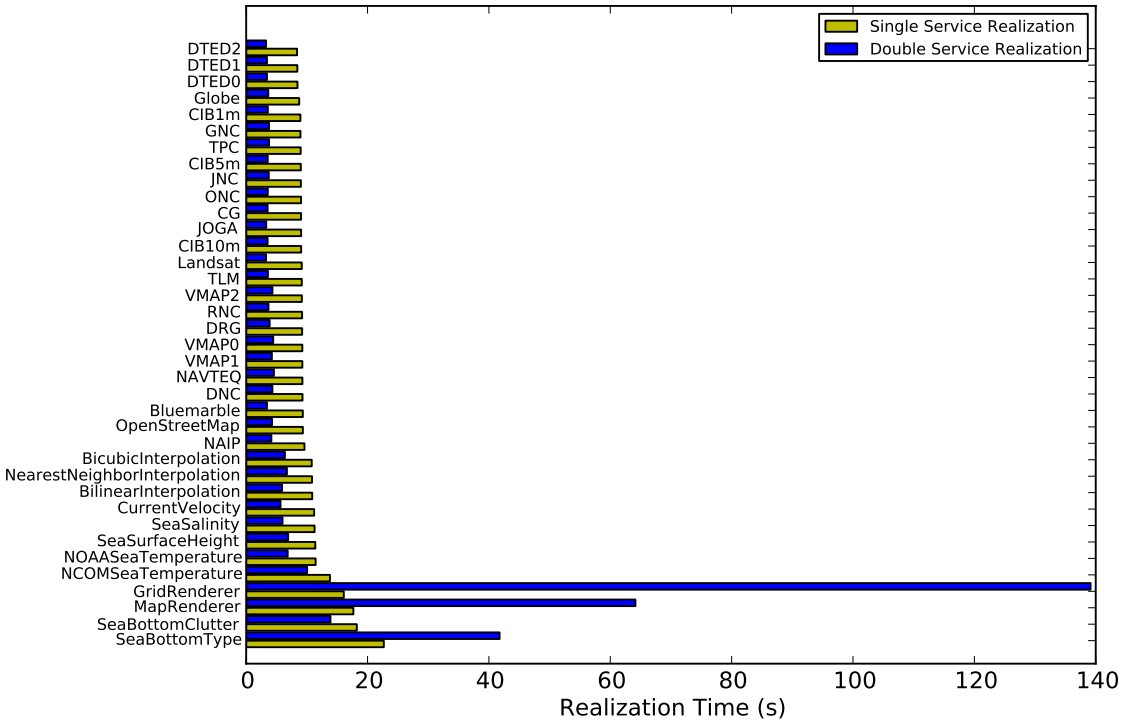


Figure B.6: Double service realization times for WindVelocity

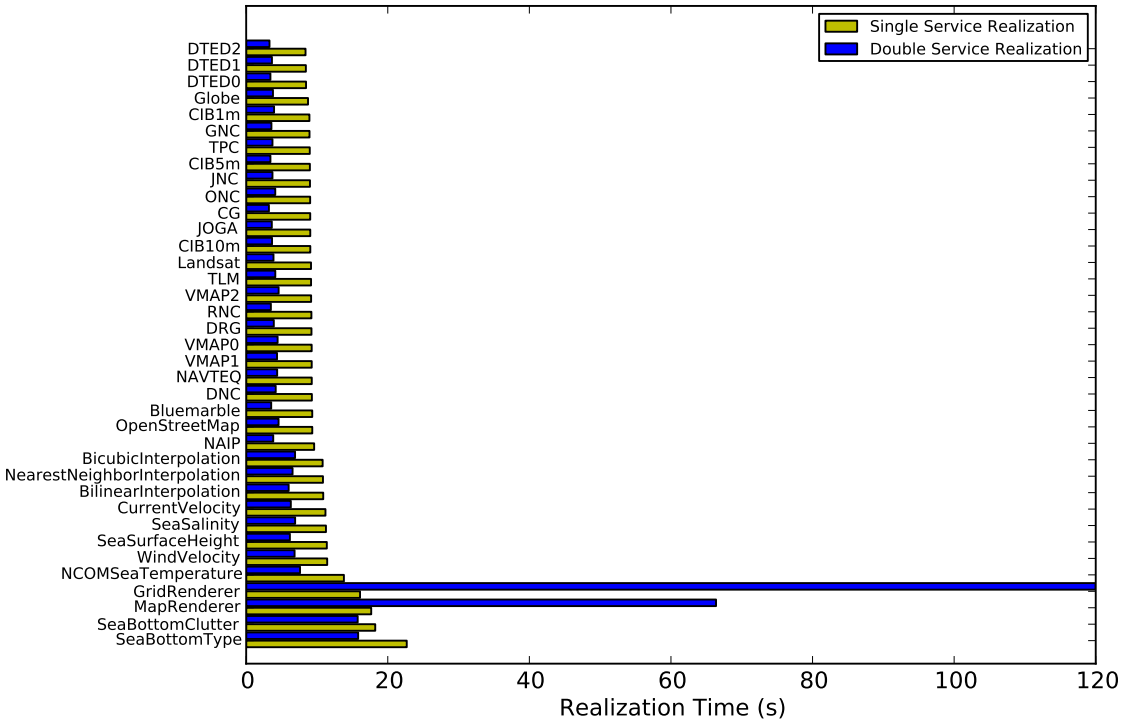


Figure B.7: Double service realization times for NOAASeaTemperature

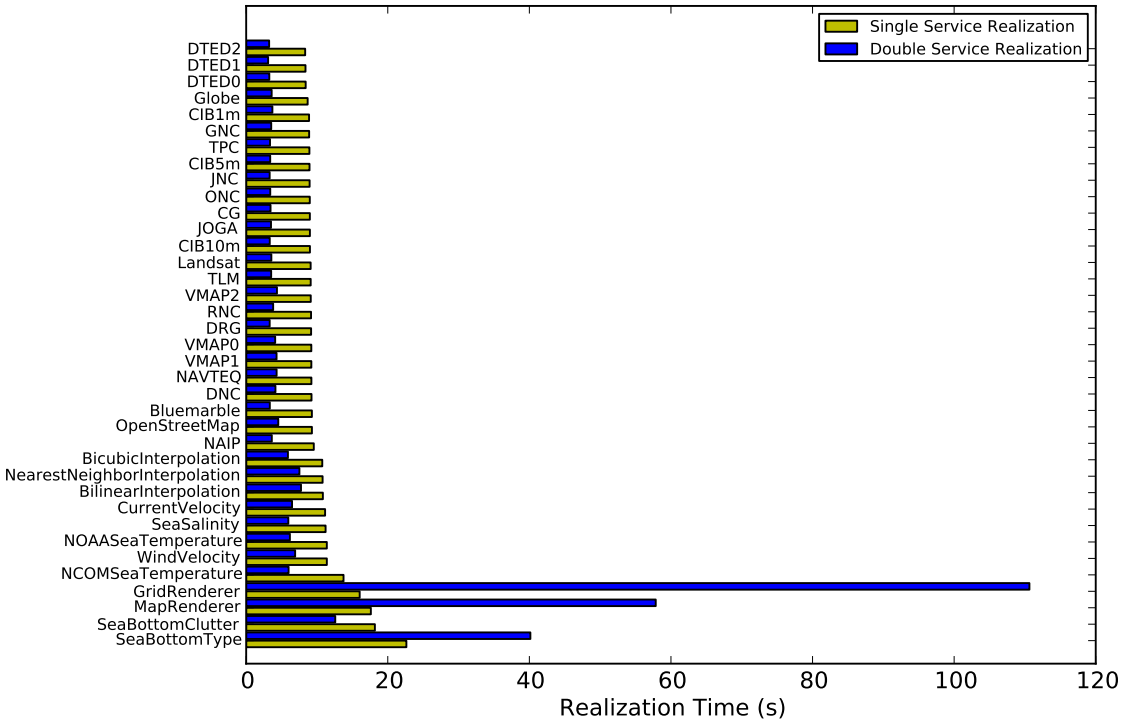


Figure B.8: Double service realization times for SeaSurfaceHeight

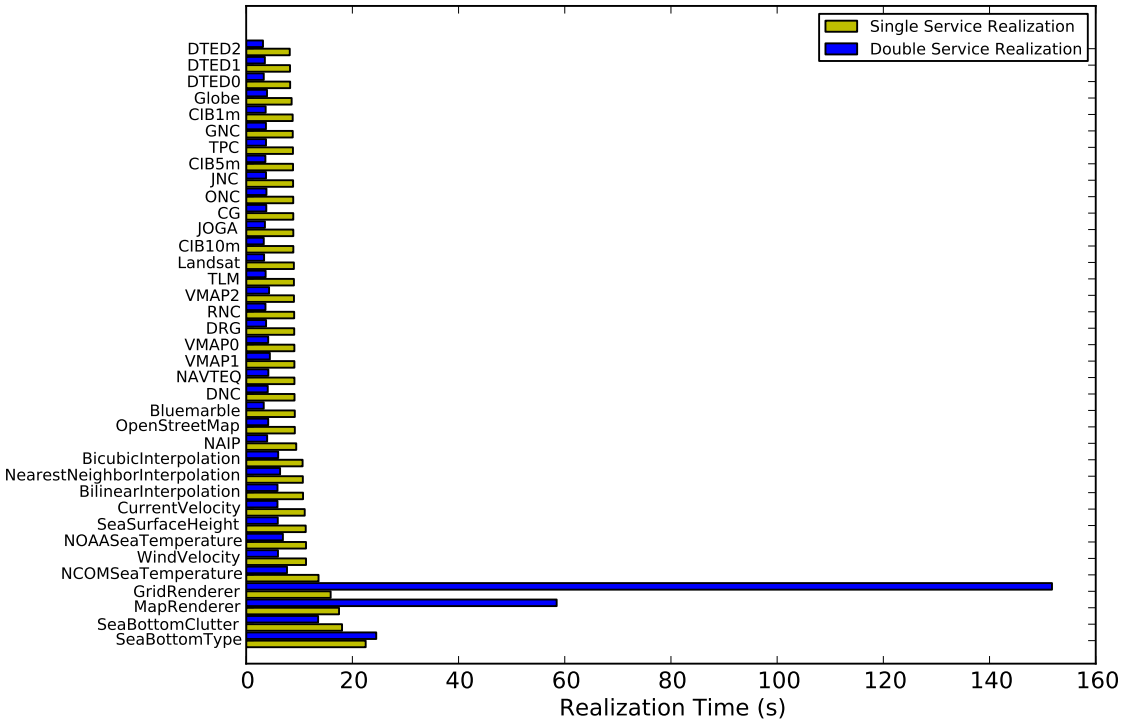


Figure B.9: Double service realization times for SeaSalinity

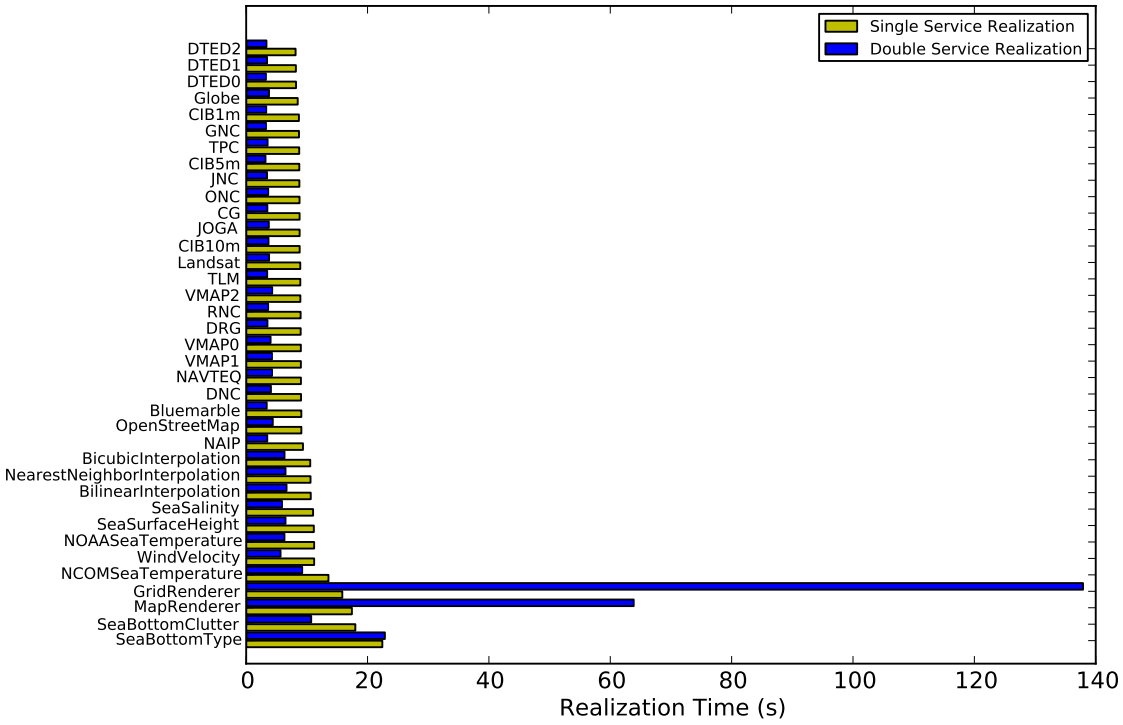


Figure B.10: Double service realization times for CurrentVelocity

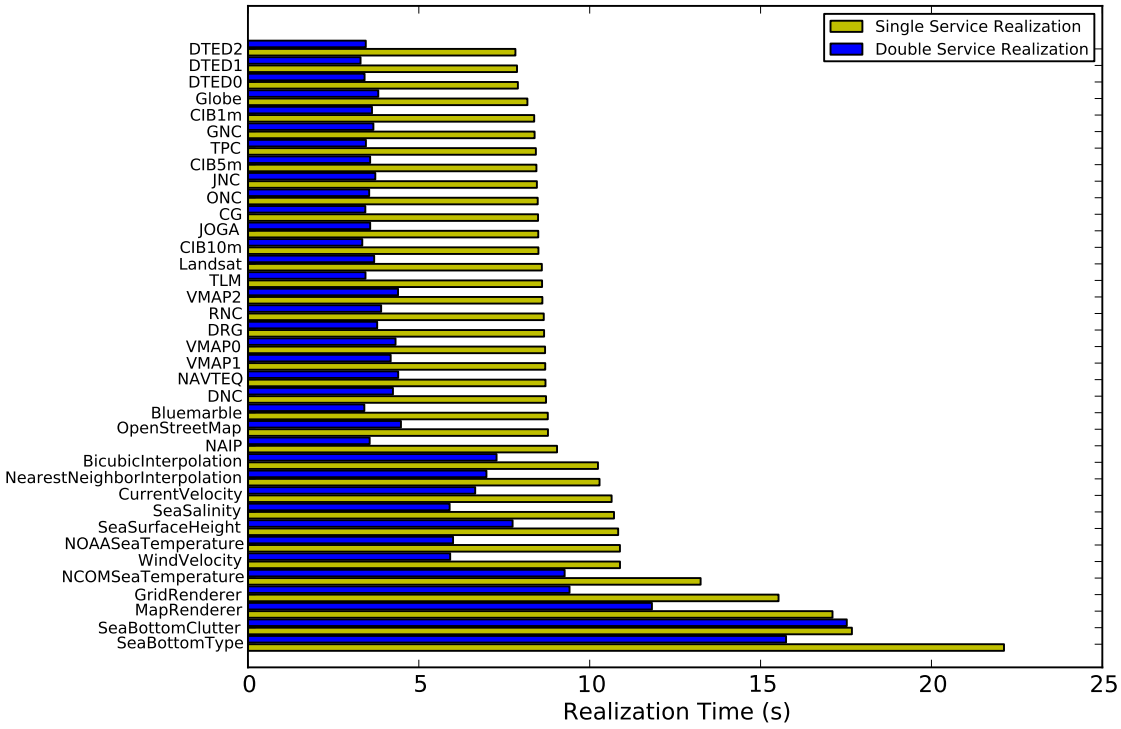


Figure B.11: Double service realization times for BilinearInterpolation

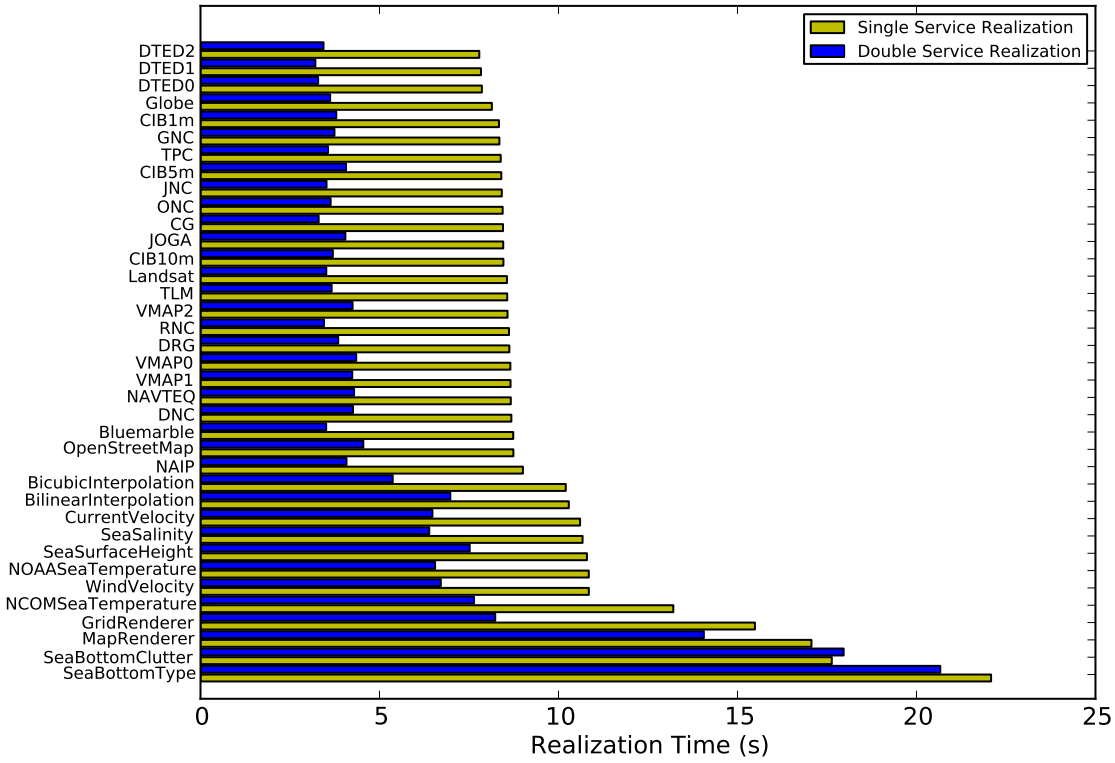


Figure B.12: Double service realization times for NearestNeighborInterpolation

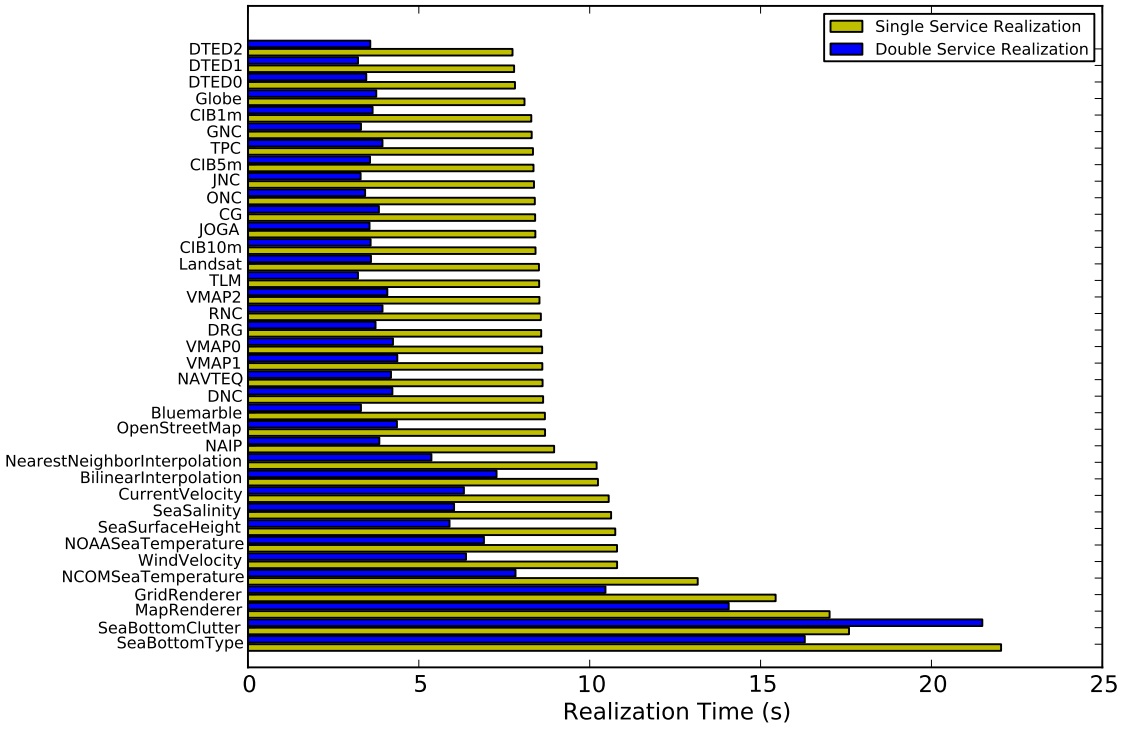


Figure B.13: Double service realization times for BicubicInterpolation

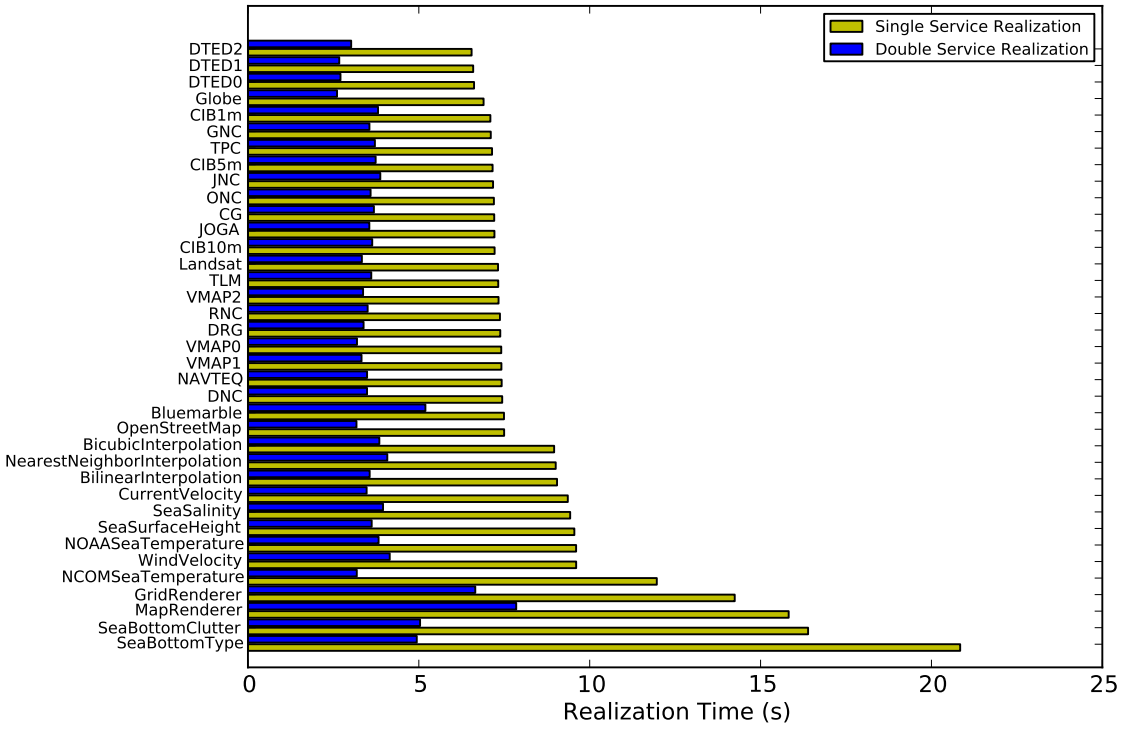


Figure B.14: Double service realization times for NAIP

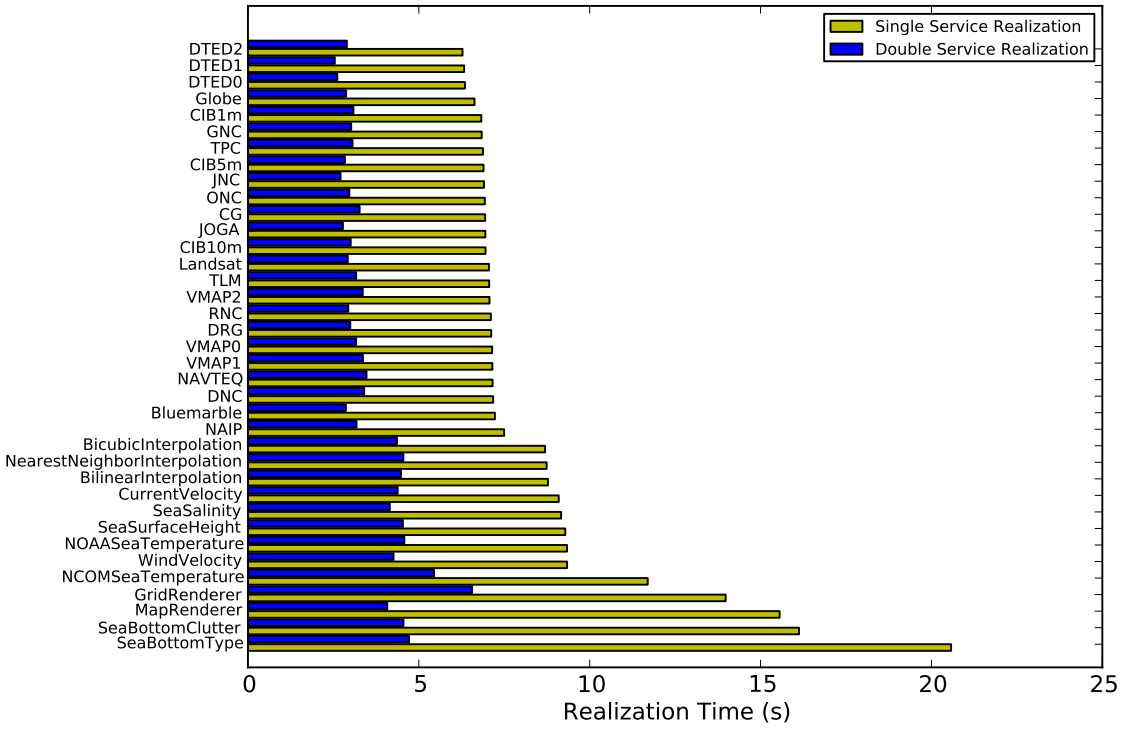


Figure B.15: Double service realization times for OpenStreetMap

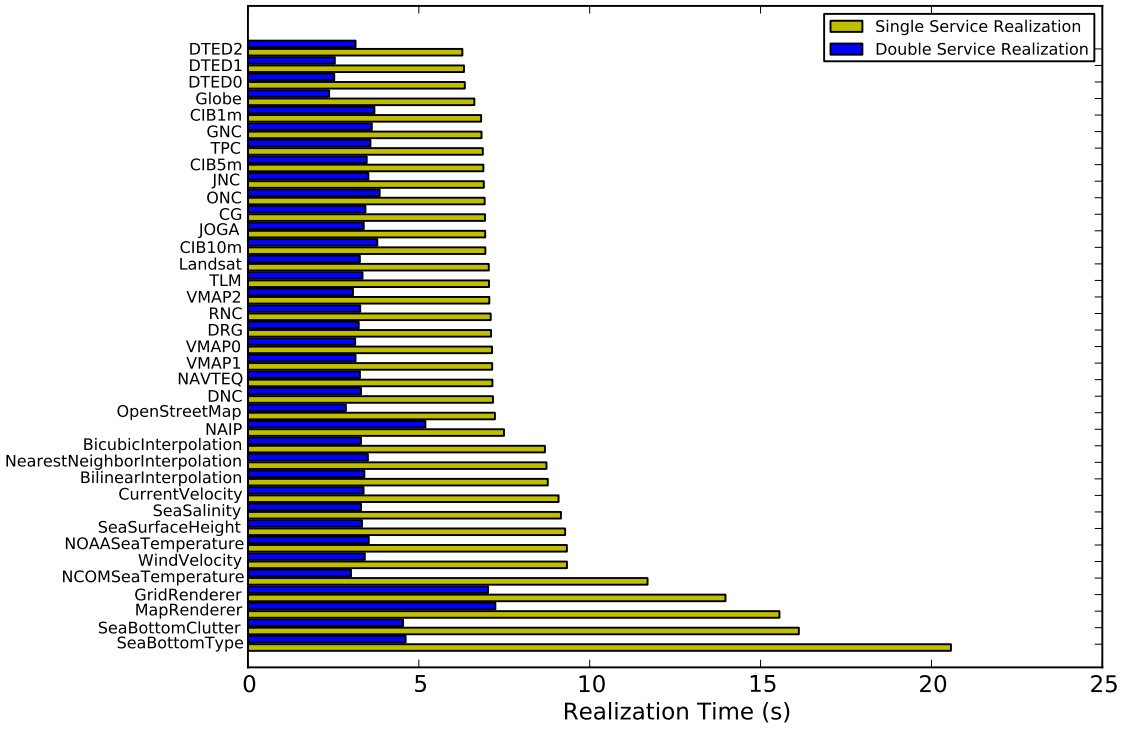


Figure B.16: Double service realization times for Bluemarble

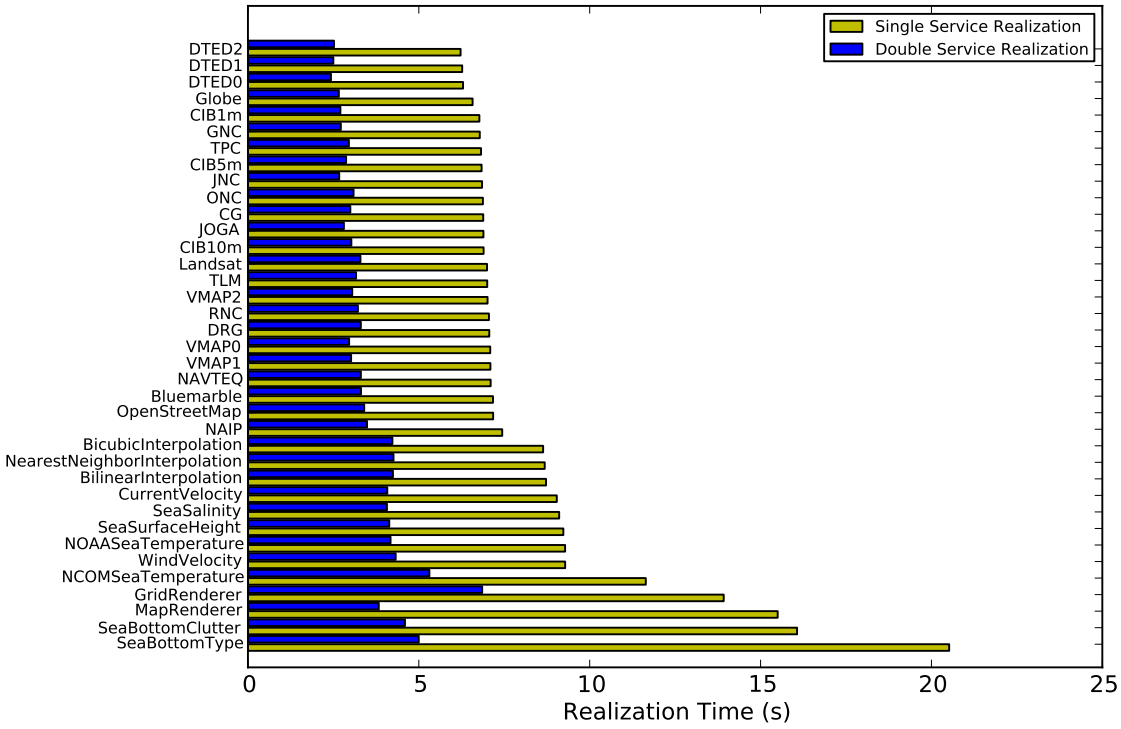


Figure B.17: Double service realization times for DNC

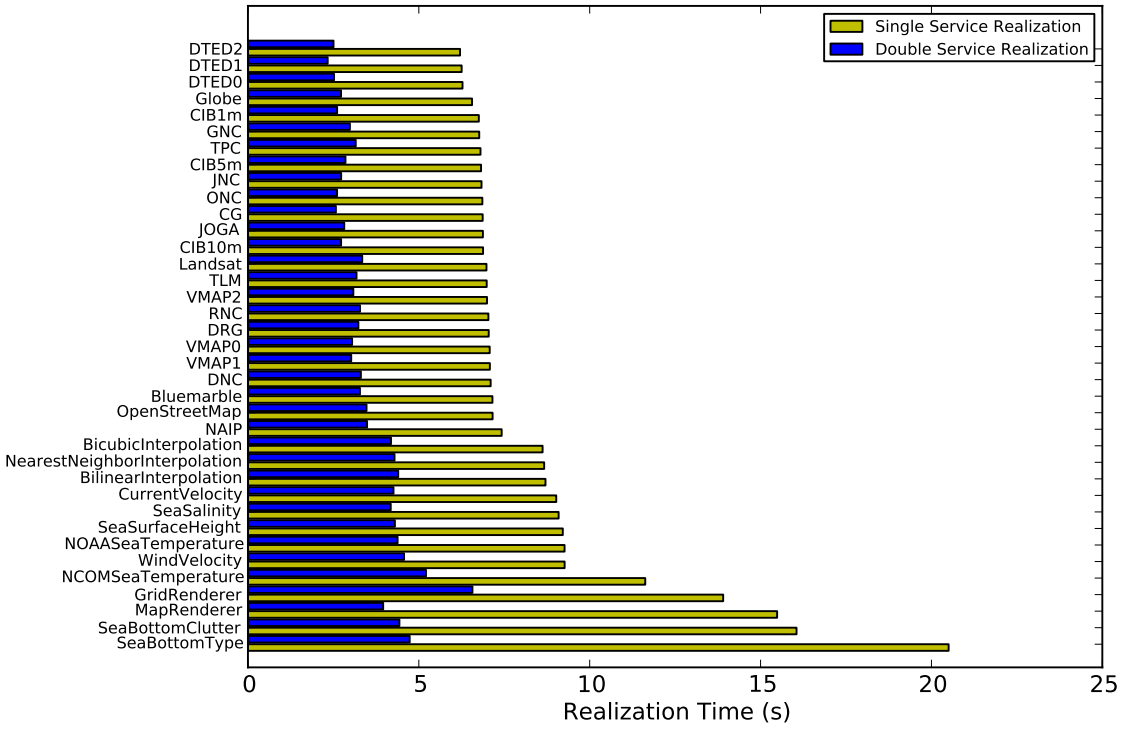


Figure B.18: Double service realization times for NAVTEQ

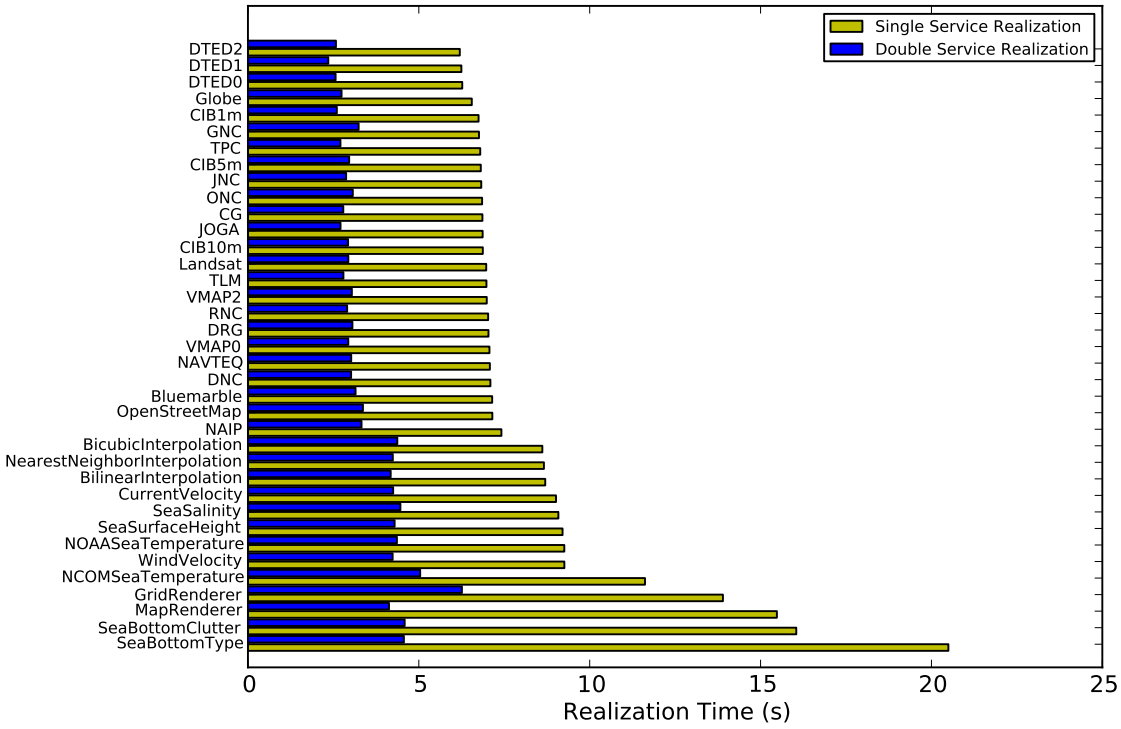


Figure B.19: Double service realization times for VMAP1

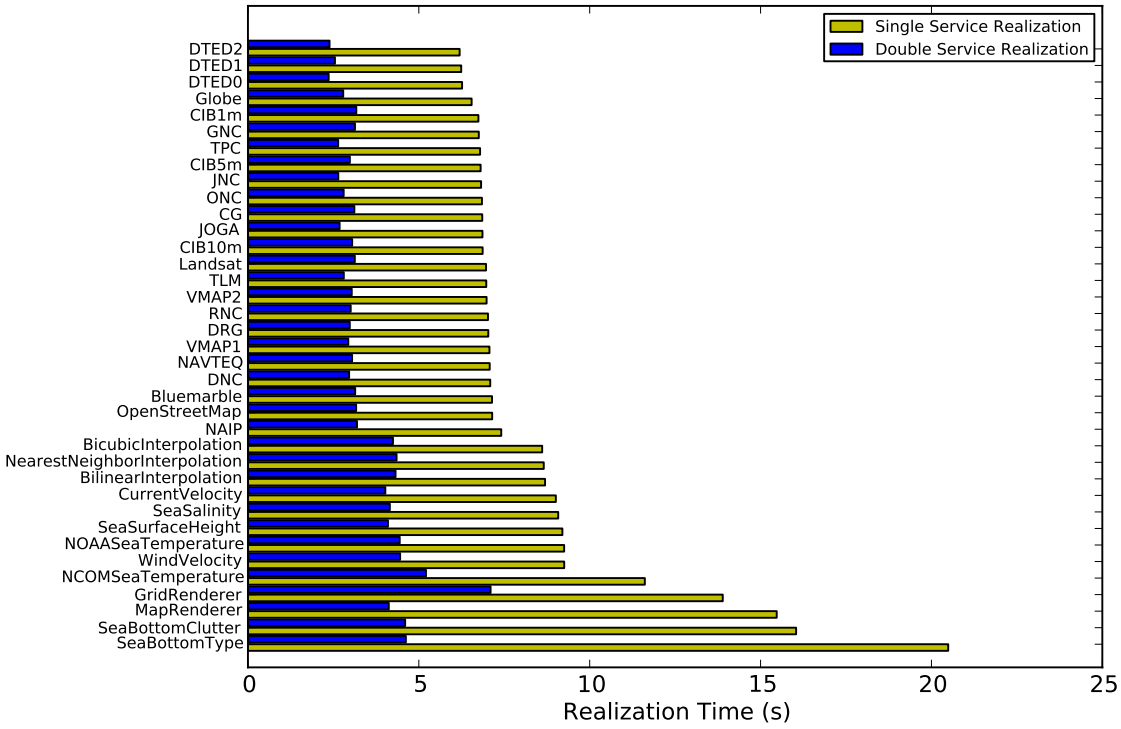


Figure B.20: Double service realization times for VMAP0

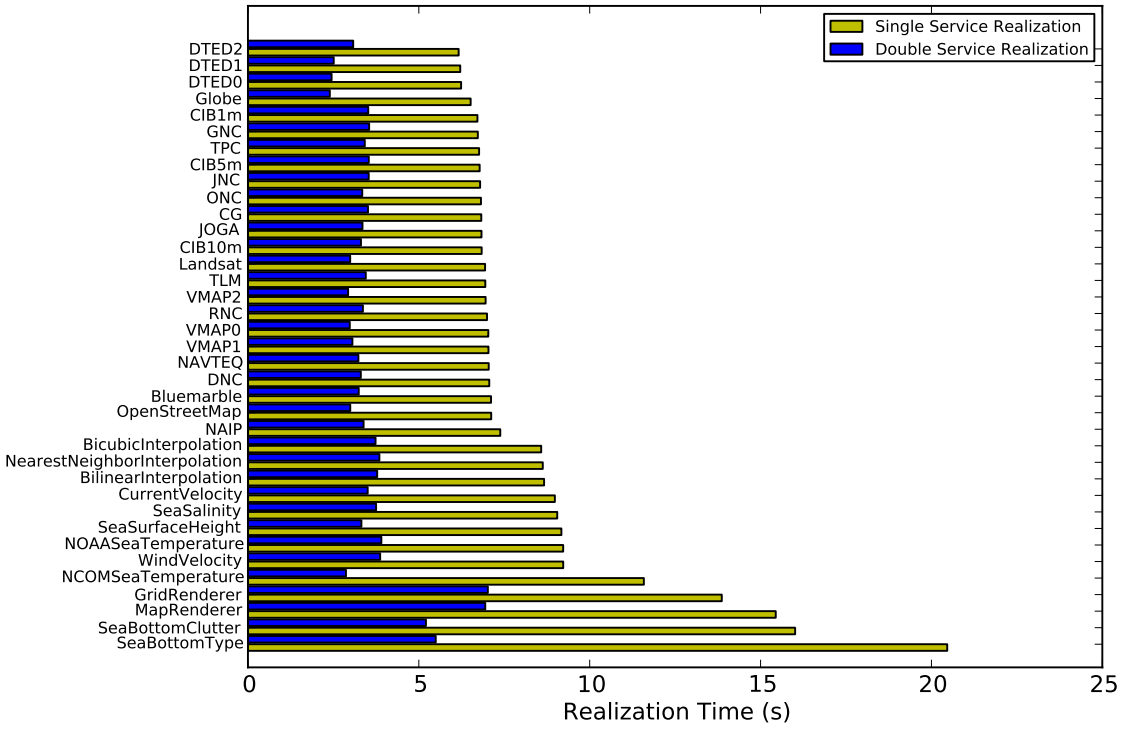


Figure B.21: Double service realization times for DRG

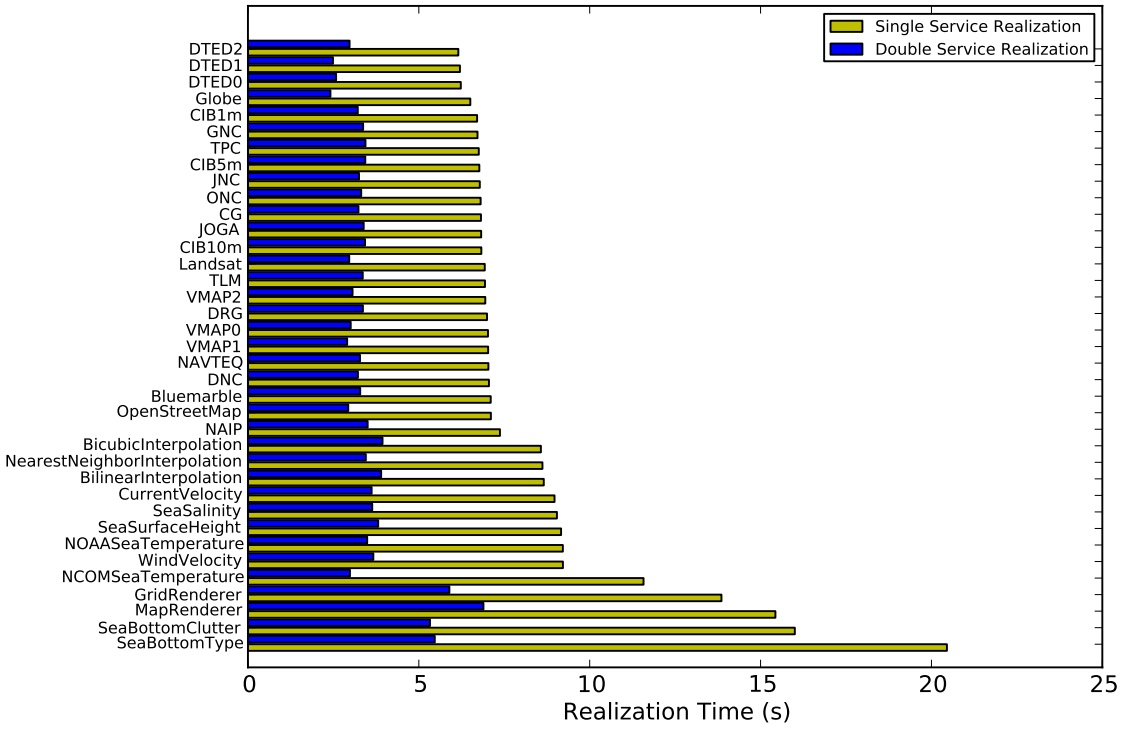


Figure B.22: Double service realization times for RNC

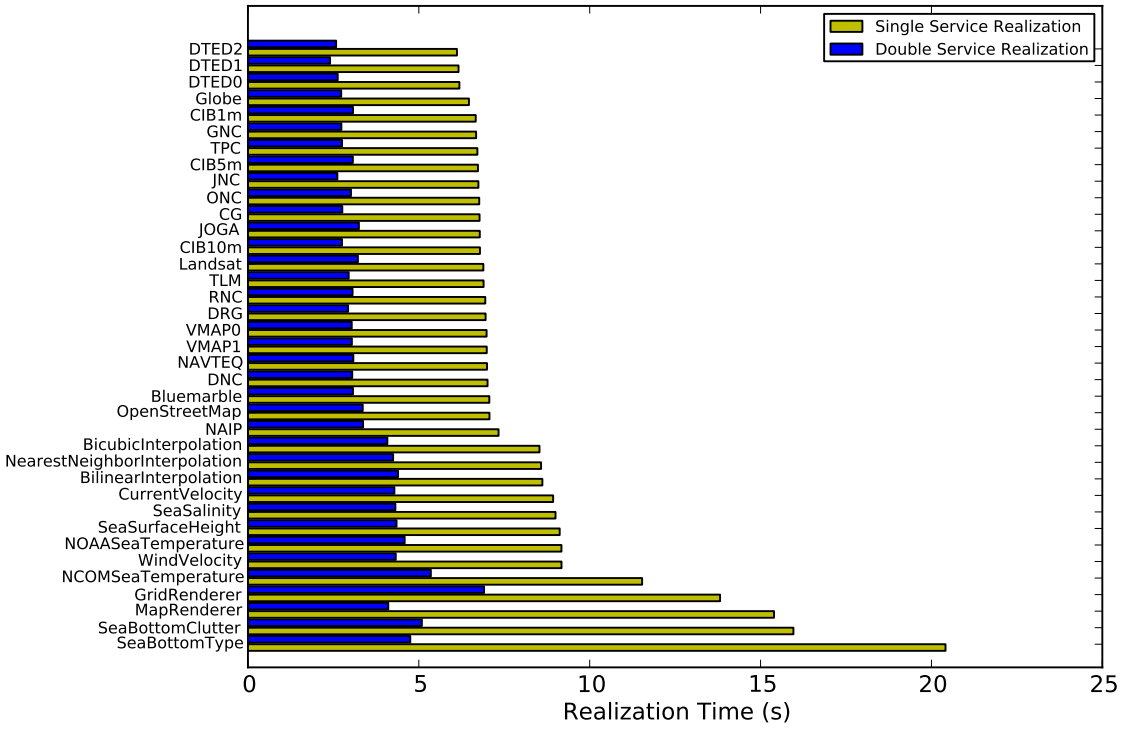


Figure B.23: Double service realization times for VMAP2

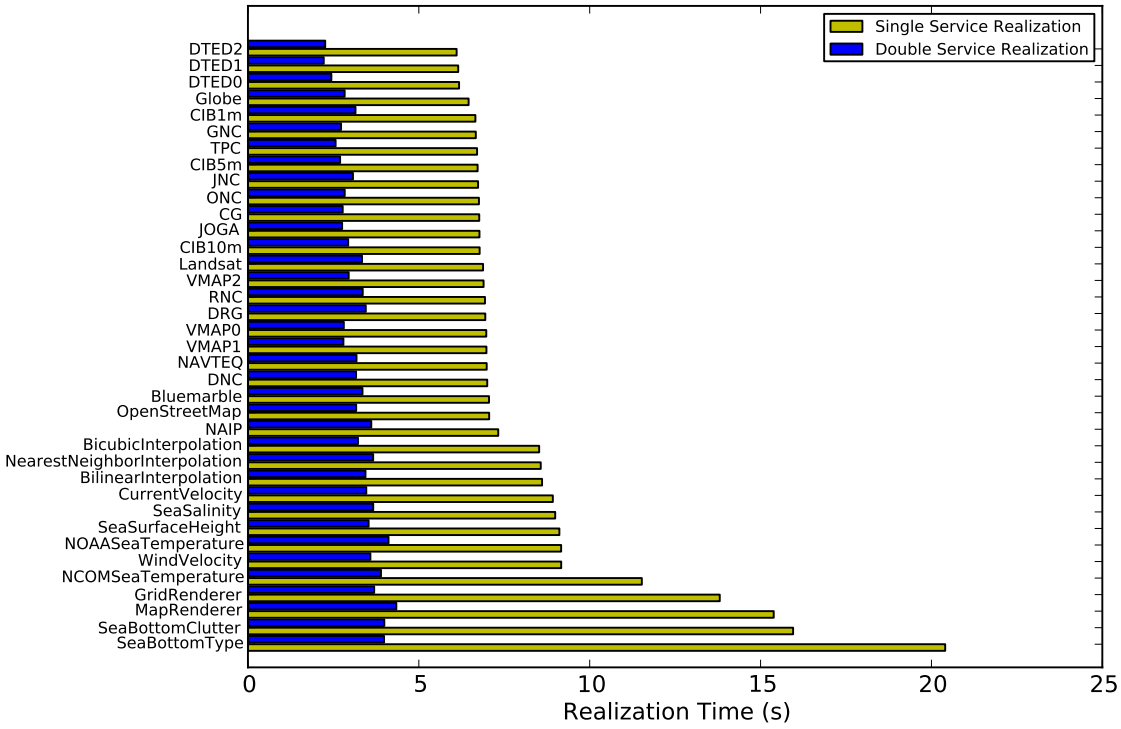


Figure B.24: Double service realization times for TLM

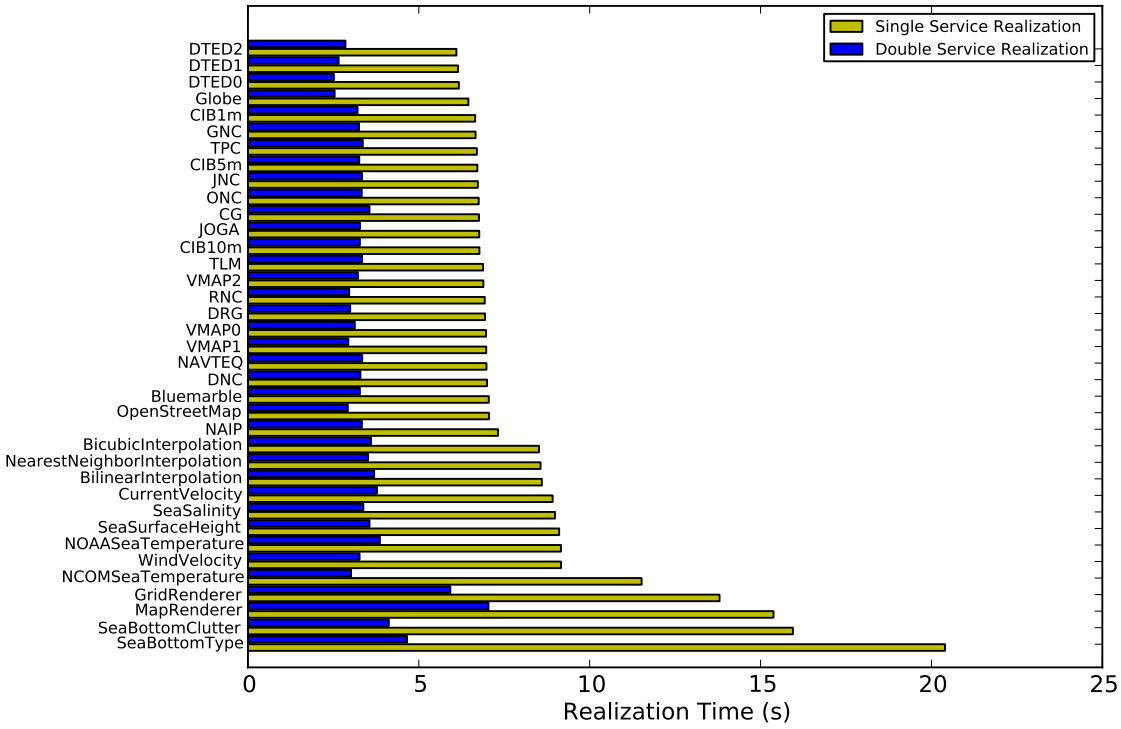


Figure B.25: Double service realization times for Landsat

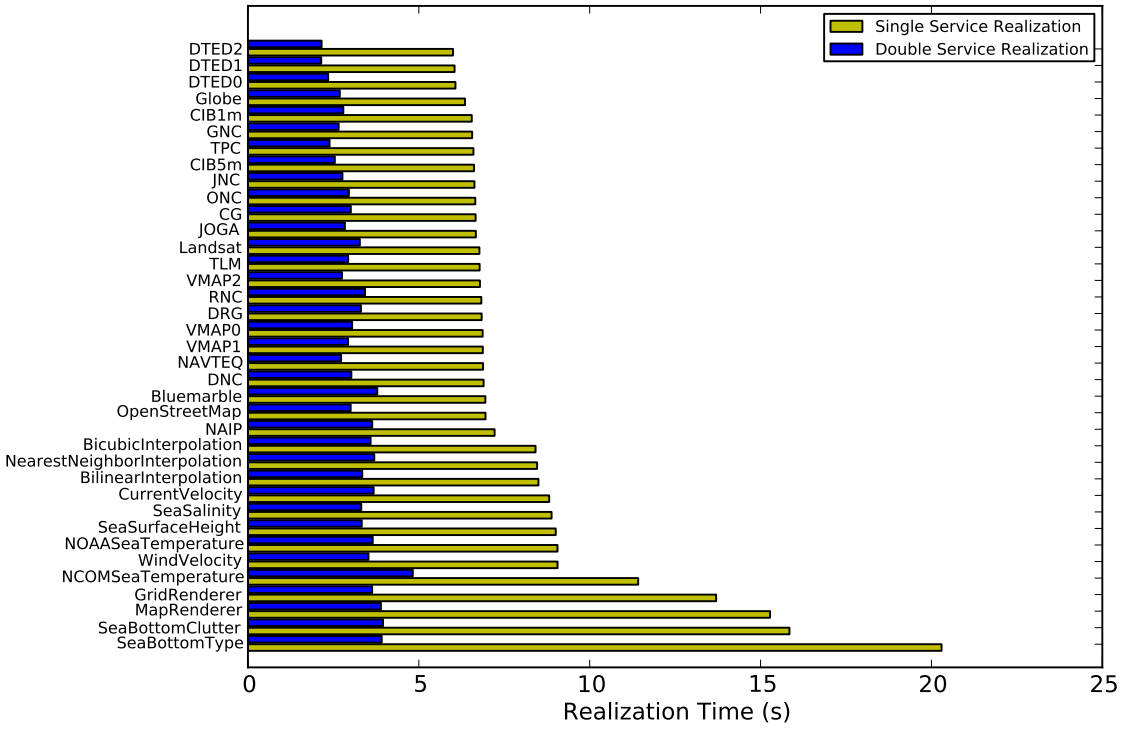


Figure B.26: Double service realization times for CIB10m

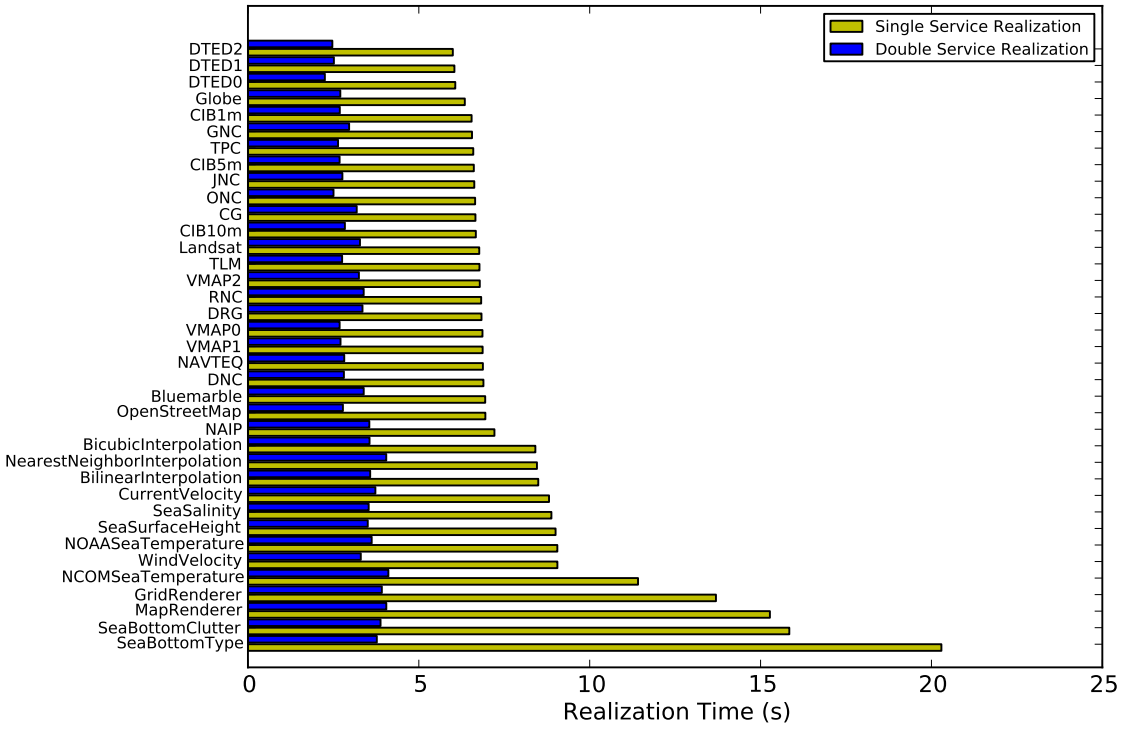


Figure B.27: Double service realization times for JOGA

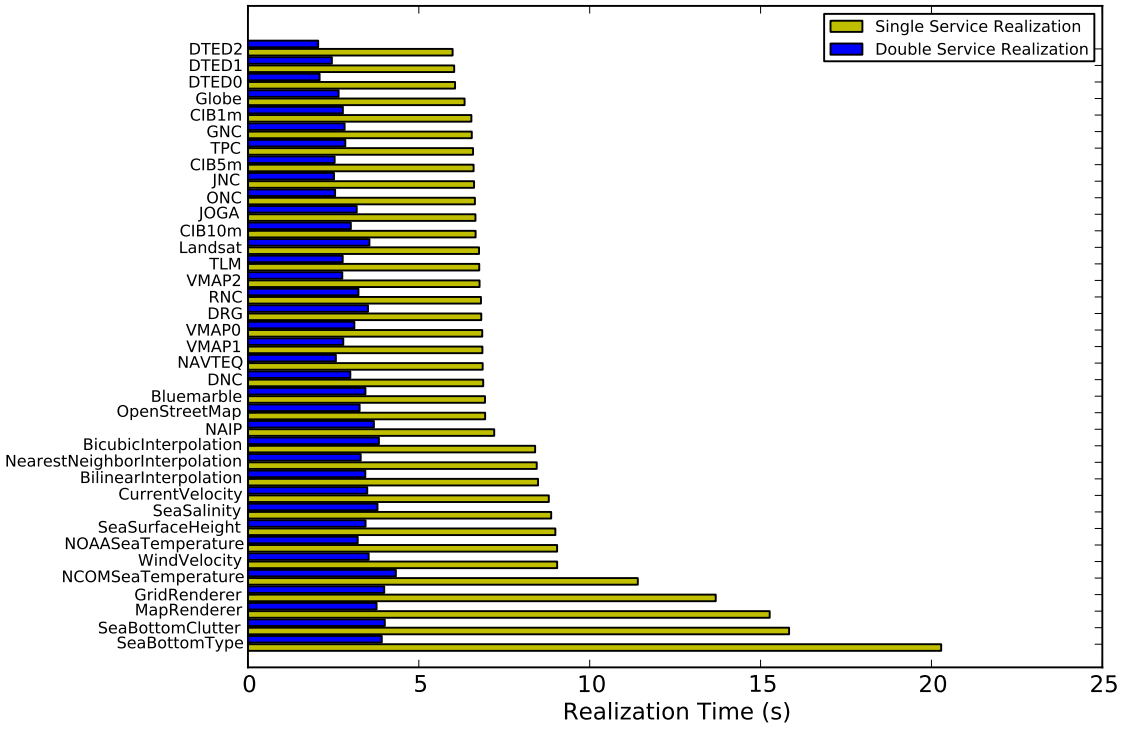


Figure B.28: Double service realization times for CG

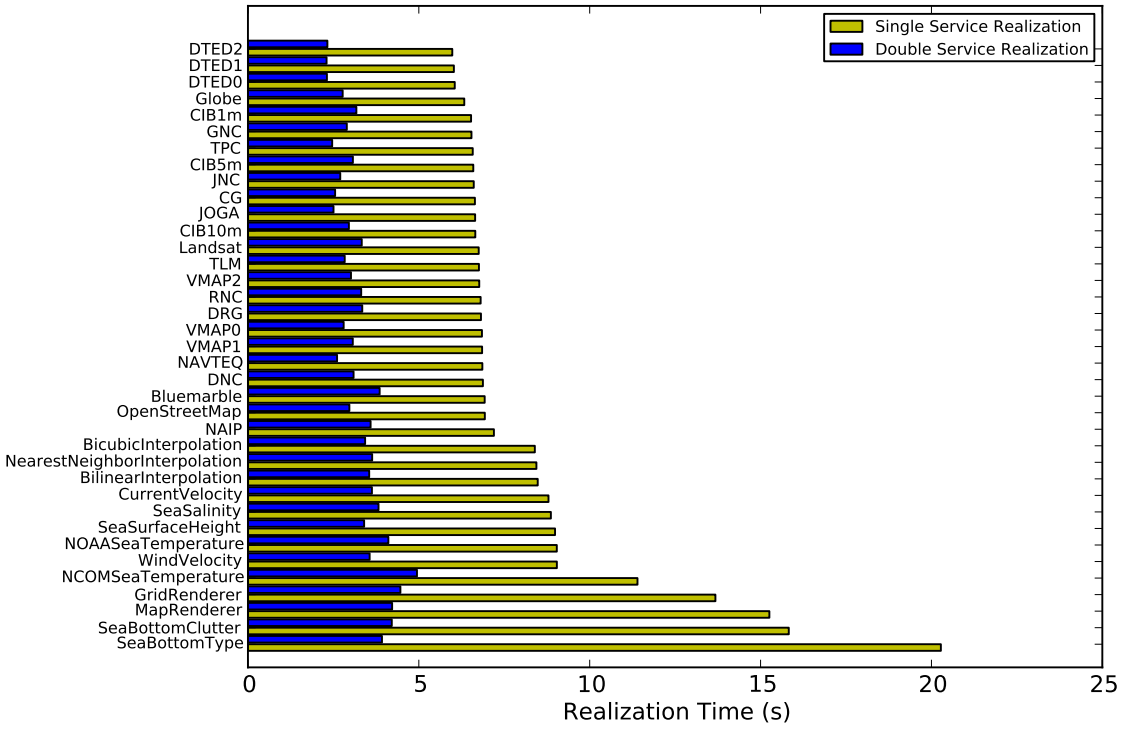


Figure B.29: Double service realization times for ONC

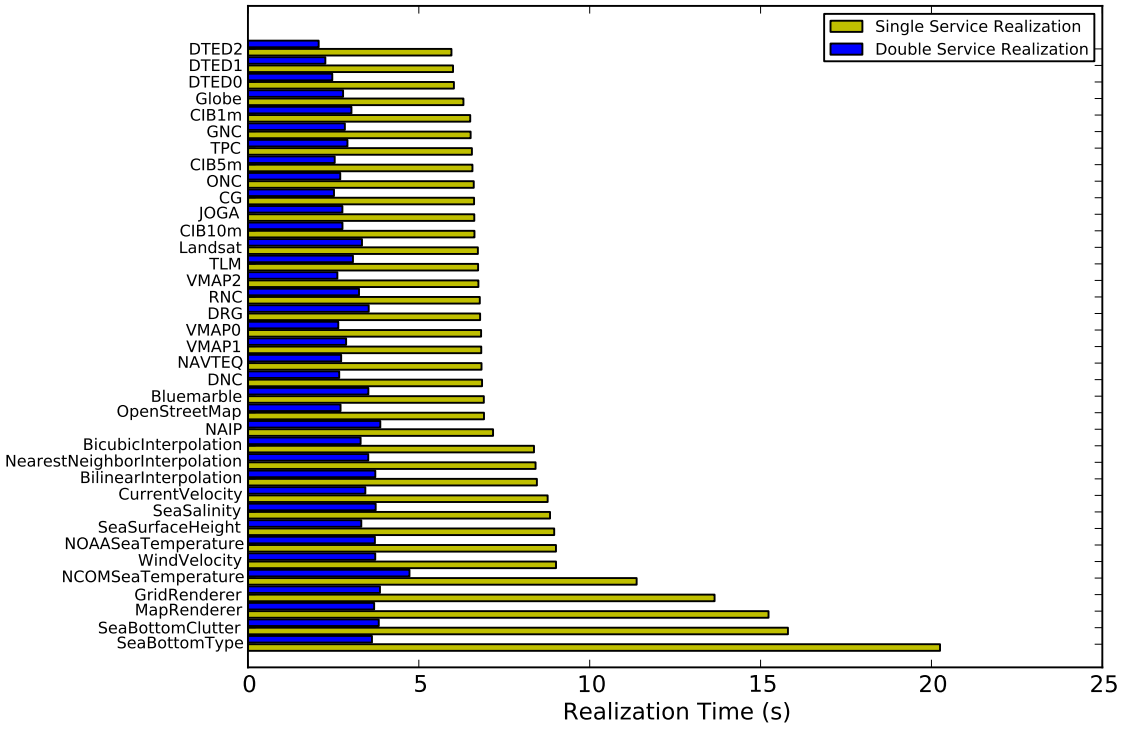


Figure B.30: Double service realization times for JNC

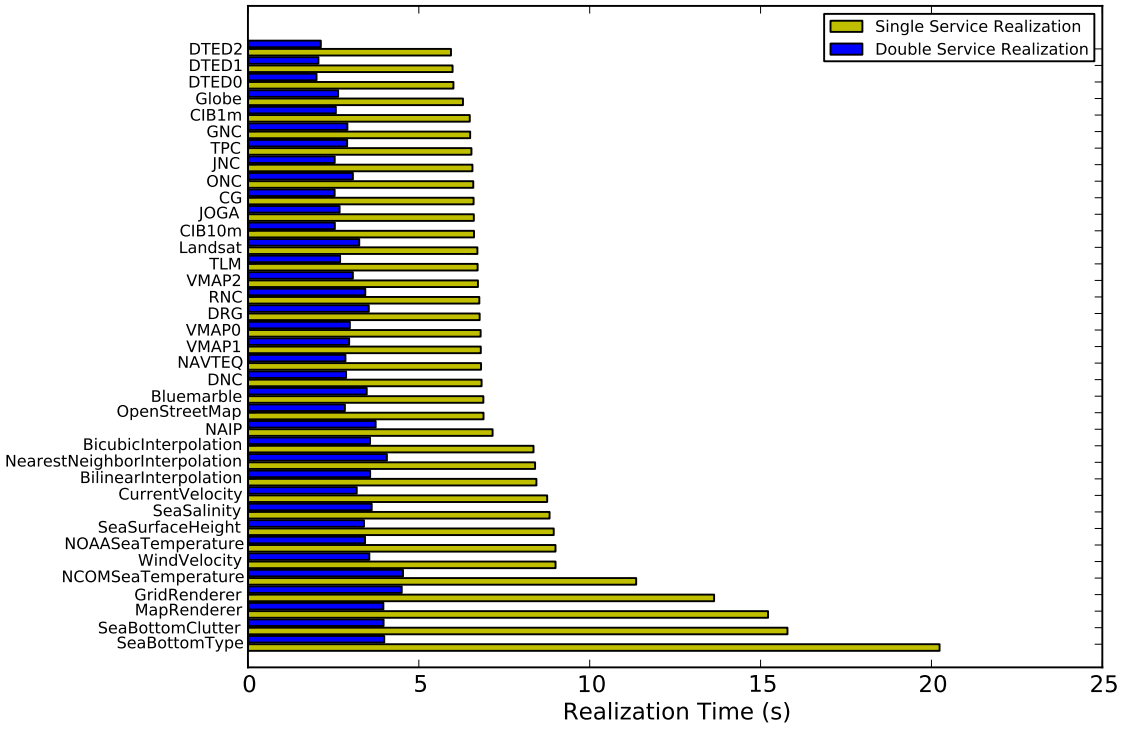


Figure B.31: Double service realization times for CIB5m

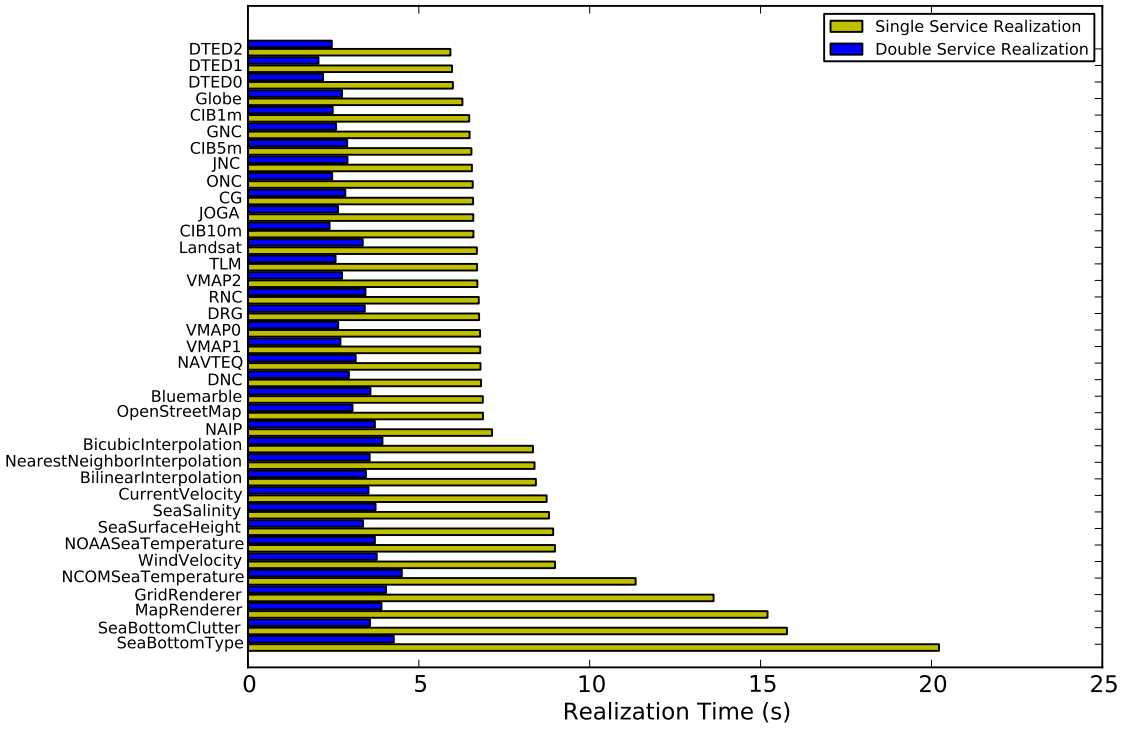


Figure B.32: Double service realization times for TPC

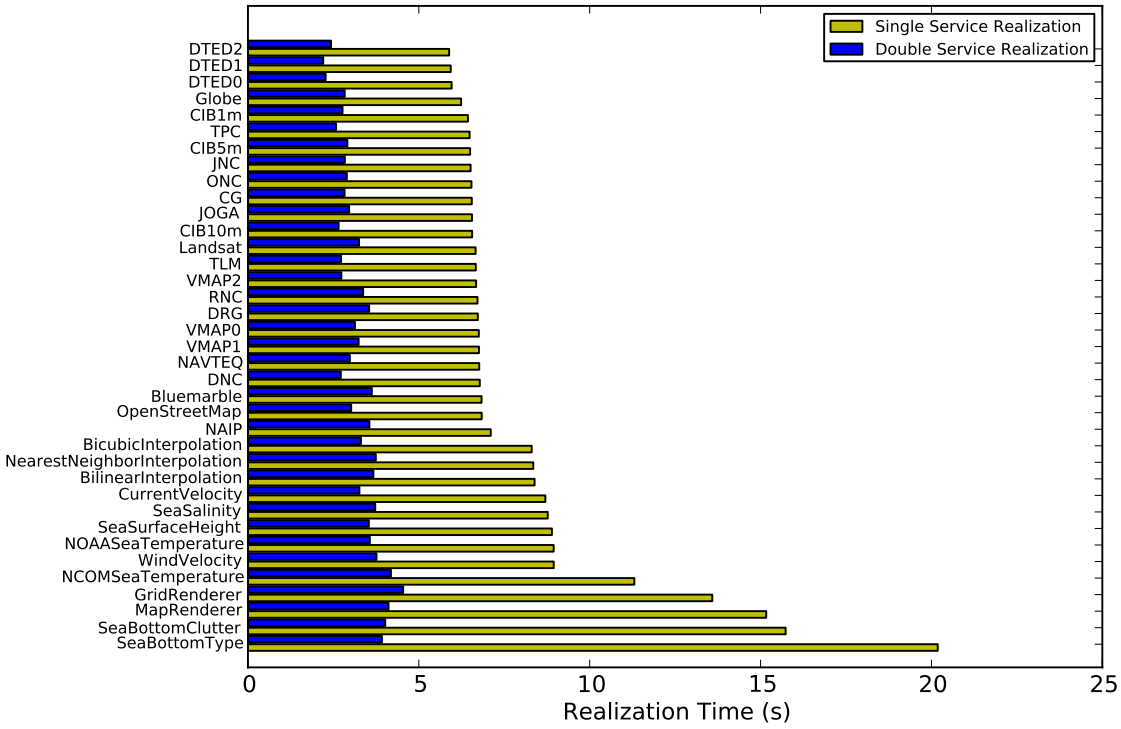


Figure B.33: Double service realization times for GNC

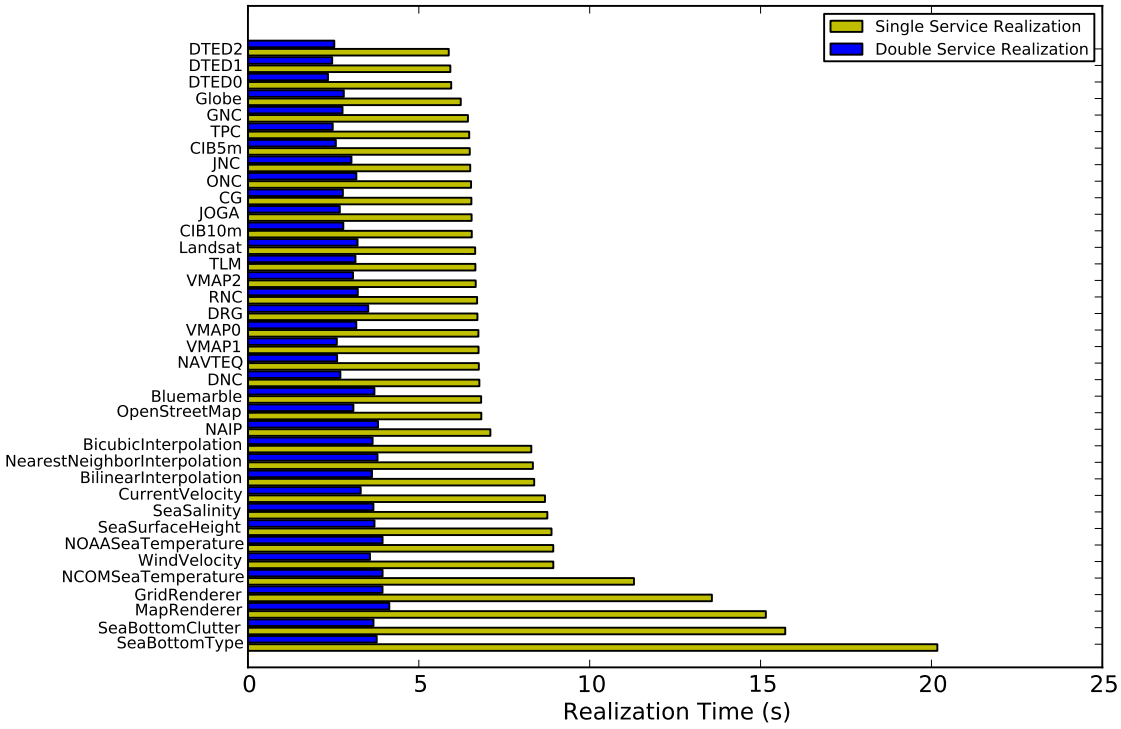


Figure B.34: Double service realization times for CIB1m

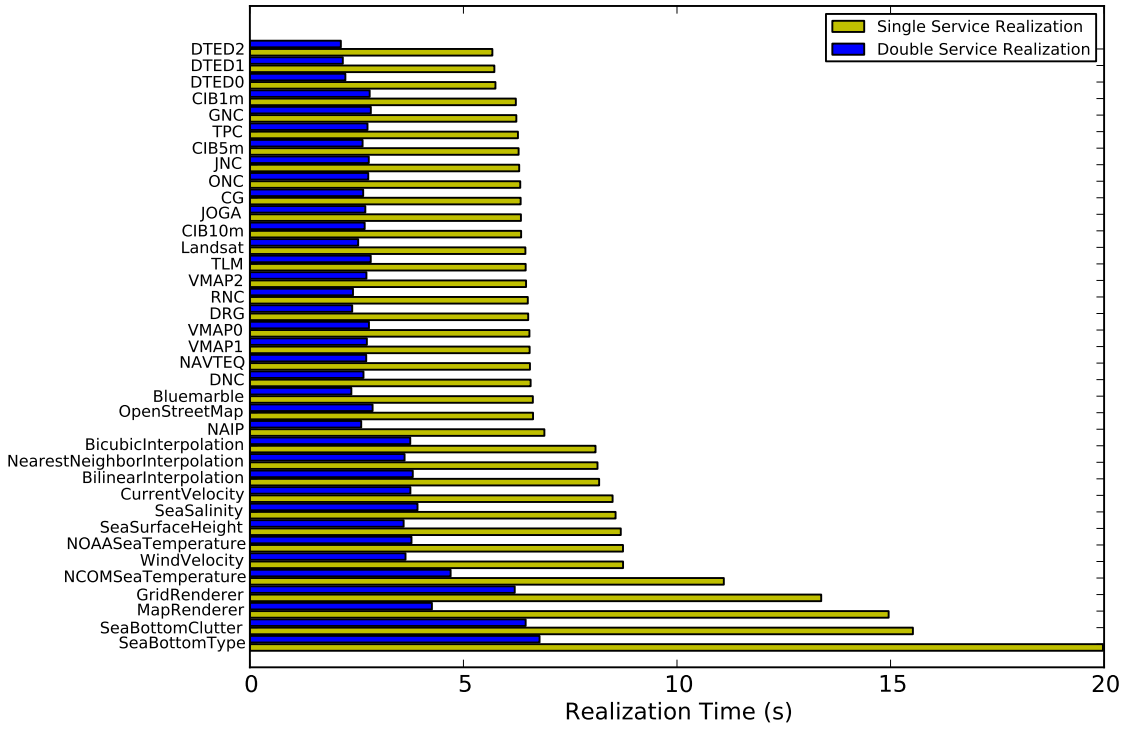


Figure B.35: Double service realization times for Globe

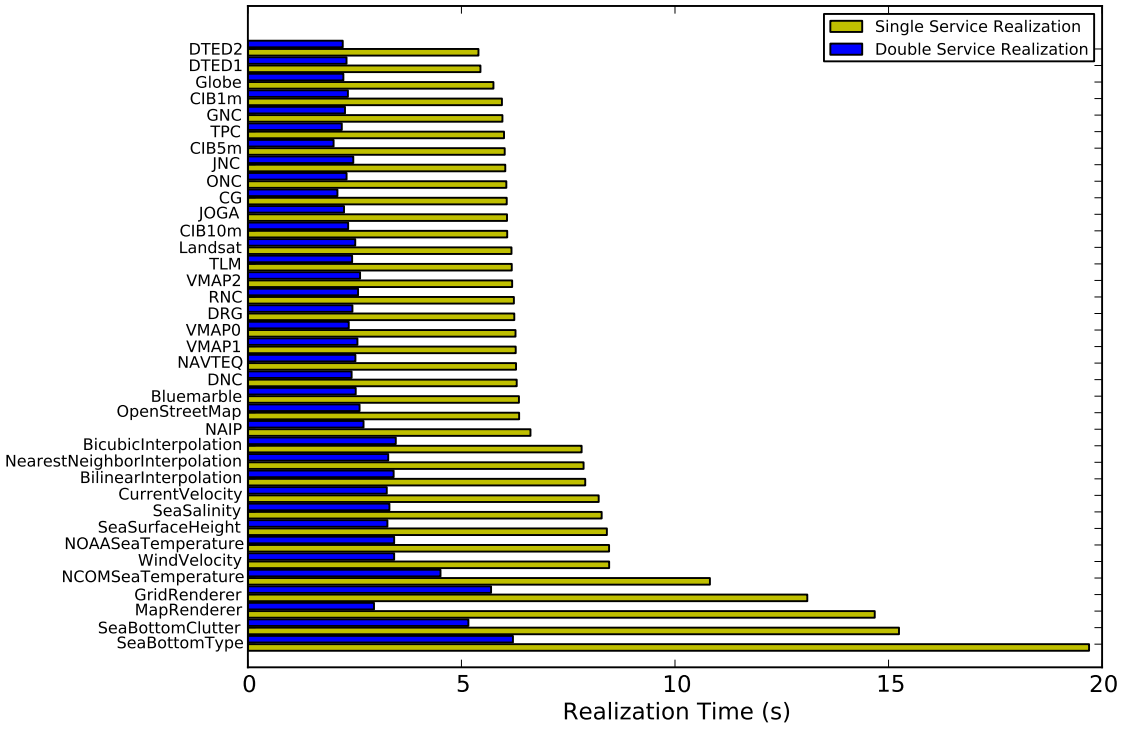


Figure B.36: Double service realization times for DTED0

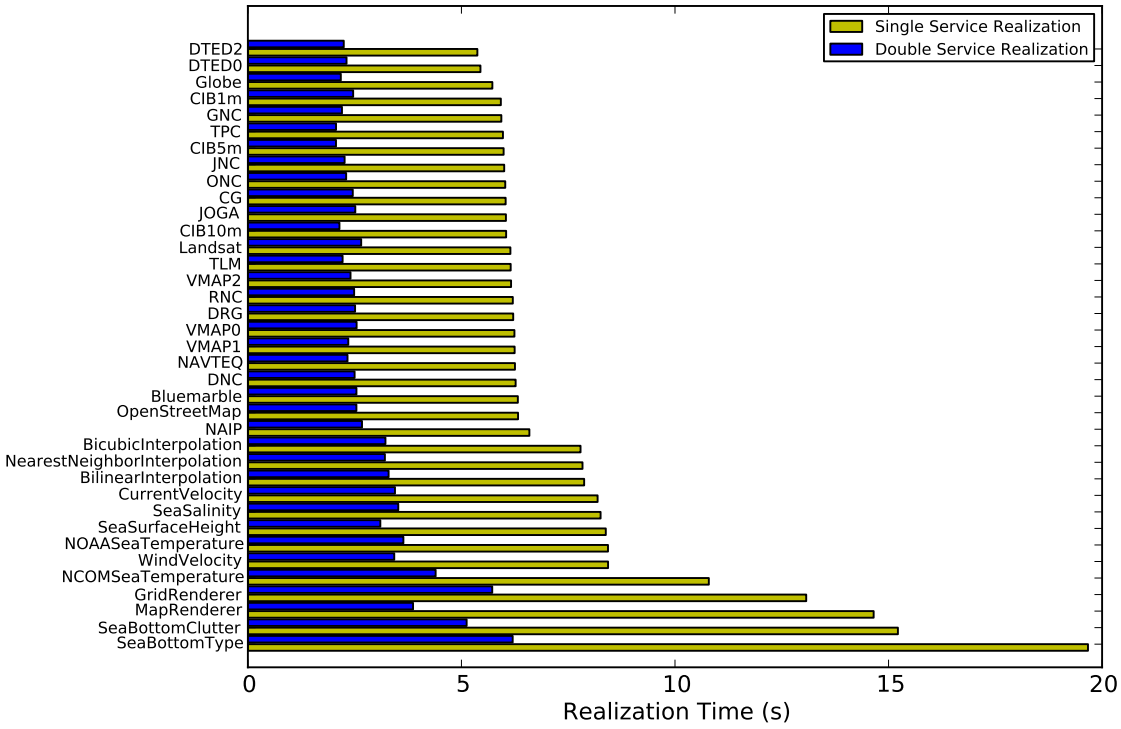


Figure B.37: Double service realization times for DTED1

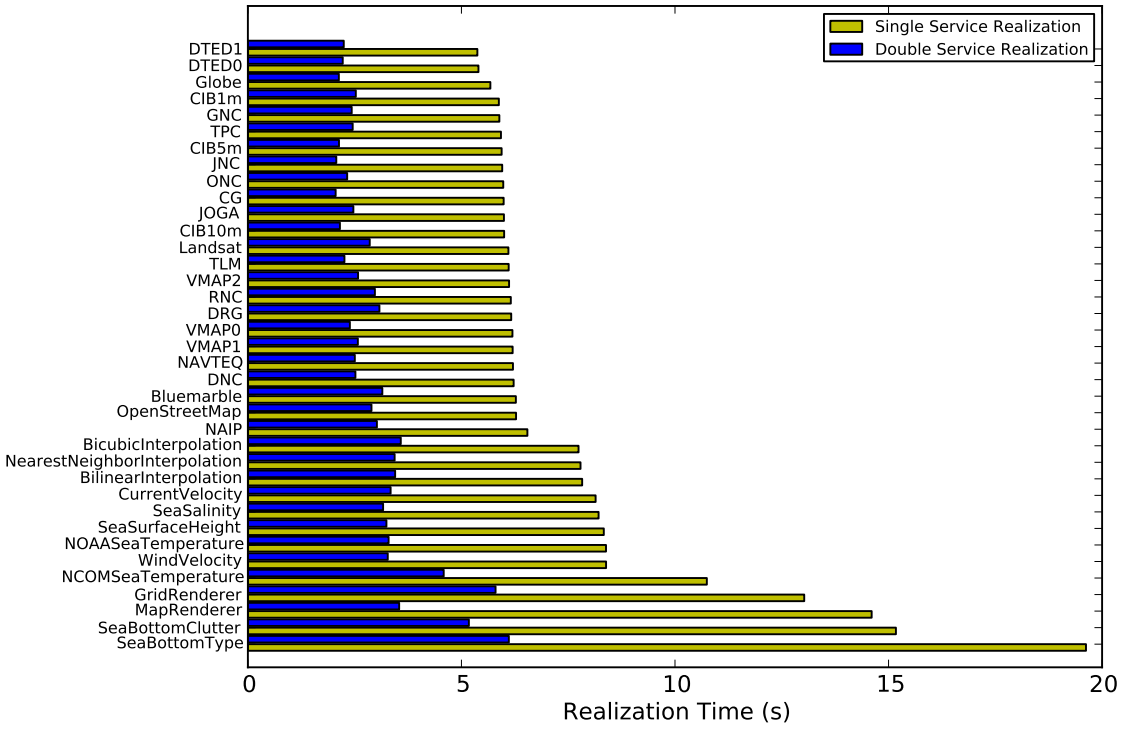


Figure B.38: Double service realization times for DTED2

Bibliography

[1] M. Aiello, C. Platzer, F. Rosenberg, H. Tran, M. Vasko, and S. Dustdar. Web service indexing for efficient retrieval and composition. In *E-Commerce Technology, 2006. The 8th IEEE International Conference on and Enterprise Computing, E-Commerce, and E-Services, The 3rd IEEE International Conference on*, pages 63–63, June 2006.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, et al. Business process execution language for web services, version 1.1. *Specification, BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems*, 2003.

[3] O. Aydın, N. Cicekli, and I. Cicekli. Automated web services composition with the event calculus. *Lecture Notes In Artificial Intelligence*, pages 142–157, 2008.

[4] F. Baader. *The description logic handbook: theory, implementation, and applications*. Cambridge Univ Pr, 2003.

[5] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web services architecture. *W3C Working Group Note*, 11:2005–1, 2004.

[6] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0. *W3C recommendation*, 6, 2000.

[7] K. Chen, J. Xu, and S. Reiff-Marganiec. Markov-htn planning approach to enhance flexibility of automatic web service composition. *Web Services, IEEE International Conference on*, 0:9–16, 2009.

[8] L. Clement, A. Hately, C. von Riegen, T. Rogers, et al. UDDI Version 3.0. 2. *UDDI Spec Technical Committee Draft, Dated*, 20041019:0–2, 2004.

[9] D. Coalition, A. Ankolekar, M. Burstein, and J. Hobbs. DAML-S: Web service description for the semantic Web. In *Proc. First International Semantic Web Conference ISWC*, pages 279–291. Springer, 2002.

[10] J. de La Beaujardière. Web map service implementation specification. *Open GIS Consortium*, 82, 2002.

[11] L. Di, P. Zhao, W. Yang, G. Yu, and P. Yue. Intelligent geospatial Web services. In *2005 IEEE International Geoscience and Remote Sensing Symposium, 2005. IGARSS'05. Proceedings*, volume 2, 2005.

[12] L. Di, P. Zhao, W. Yang, and P. Yue. Ontology-driven Automatic Geospatial-Processing Modeling based on Web-service Chaining. In *Proceedings of the Sixth Annual NASA Earth Science Technology Conference*. Citeseer, 2006.

[13] J. Evans. Web Coverage Service (WCS), Version 1.0.0. *Open Geospatial Consortium, Wayland, MA, USA*, 2003.

[14] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol-HTTP/1.1., 1999.

[15] K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to Web services architecture. *IBM SYSTEMS JOURNAL*, 41(2), 2002.

[16] I. Horrocks. DAML+ OIL: a description logic for the semantic web. *Bulletin of the Technical Committee on*, 51:4, 2002.

[17] E. Ioup, B. Lin, J. Sample, and K. Shaw. Geospatial Web Services: Bridging the Gap between OGC and Web Services. In M. A. John Sample, Shengru Tu, editor, *Geospatial Services and Applications for the Internet*, chapter 4, pages 72–92. Springer, 2008.

[18] S. Kalasapur, M. Kumar, and B. Shirazi. Dynamic service composition in pervasive computing. *IEEE Transactions on Parallel and Distributed Systems*, pages 907–918, 2007.

[19] M. Klein, B. Konig-Ries, and M. Mussig. What is needed for semantic service descriptions? A proposal for suitable language constructs. *International Journal of Web and Grid Services*, 1(3):328–364, 2005.

[20] S. Kona, A. Bansal, M. Blake, and G. Gupta. Towards a general framework for web service composition. In *Services Computing, 2008. SCC '08. IEEE International Conference on*, volume 2, pages 497–498, July 2008.

[21] J. Kopeckỳ, T. Vitvar, C. Bournez, and J. Farrell. SAWSDL: Semantic Annotations for WSDL and XML Schema. *IEEE Internet Computing*, 11(6):60–67, 2007.

[22] U. Küster, B. König-Ries, M. Stern, and M. Klein. DIANE: an integrated approach to automated service discovery, matchmaking and composition. In *Proceedings of the 16th international conference on World Wide Web*, pages 1033–1042. ACM Press New York, NY, USA, 2007.

[23] J.-J. Le and F. He. Automatic web services composition based on reasoning petri net. In *Advanced Language Processing and Web Information Technology, 2008. ALPIT '08. International Conference on*, pages 569–574, July 2008.

[24] Q. Liang and H. Lam. Web service matching by ontology instance categorization. In *Services Computing, 2008. SCC '08. IEEE International Conference on*, volume 1, pages 202–209, July 2008.

[25] J. Lieberman, T. Pehle, C. Morris, D. Kolas, M. Dean, M. Lutz, F. Probst, and E. Klien. Geospatial semantic web interoperability experiment report. *Wayland, MA, Open Geospatial Consortium Technical Report*, 2006.

[26] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, et al. OWL-S: Semantic markup for web services, 2004.

[27] D. McGuinness, F. Van Harmelen, et al. OWL web ontology language overview. *W3C recommendation*, 10:2004–03, 2004.

[28] L. McKee. The Importance of Going "Open". *Open Geospatial Consortium*, 2003.

[29] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th international conference on World Wide Web*, pages 77–88. ACM New York, NY, USA, 2002.

[30] I. Paik and D. Maruyama. Automatic Web Services Composition Using Combining HTN and CSP. In *7th IEEE International Conference on Computer and Information Technology, 2007. CIT 2007*, pages 206–211, 2007.

[31] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 333–347, 2002.

[32] C. Peltz. Web services orchestration and choreography. *Computer*, pages 46–52, 2003.

[33] J. Rao, P. Kungas, and M. Matskin. Logic-based Web services composition: from service description to process model. In *Web Services, 2004. Proceedings. IEEE International Conference on*, pages 446–453, 2004.

[34] R. Raskin and M. Pan. Knowledge representation in the semantic web for Earth and environmental terminology (SWEET). *Computers and Geosciences*, 31(9):1119–1125, 2005.

[35] K. Ren, X. Liu, J. Chen, N. Xiao, J. Song, and W. Zhang. A QSQL-based efficient planning algorithm for fully-automated service composition in dynamic service environments. *Services Computing, 2008. SCC '08. IEEE International Conference on*, 1:301–308, July 2008.

[36] K. Ren, J. Song, J. Chen, N. Xiao, and C. Liu. A Pre-reasoning Based Method for Service Discovery and Service Instance Selection in Service Grid Environments. In *Asia-Pacific Service Computing Conference, The 2nd IEEE*, pages 320–327, 2007.

[37] G. Salaun, L. Bordeaux, and M. Schaerf. Describing and reasoning on Web Services using Process Algebra. *International Journal of Business Process Integration and Management*, 1(2):116–128, 2006.

[38] K. Schild. A Correspondence Theory for Terminological Logics: Preliminary Report. In *In Proc. of IJCAI-91*. Citeseer, 1991.

[39] A. Sheth, C. Henson, and S. Sahoo. Semantic sensor web. *IEEE Internet Computing*, pages 78–83, 2008.

[40] M. Shiaa, J. Fladmark, and B. Thiell. An incremental graph-based approach to automatic service composition. In *Services Computing, 2008. SCC '08. IEEE International Conference on*, volume 1, pages 397–404, July 2008.

[41] M. Singh and M. Huhns. *Service-oriented computing: semantics, processes, agents*. John Wiley & Sons Inc, 2005.

[42] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic composition of web services using semantic descriptions. In *Web Services: Modeling, Architecture and Infrastructure" workshop in conjunction with ICEIS2003*, 2002.

[43] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN planning for web service composition using SHOP2. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):377–396, 2004.

[44] K. Stock, A. Robertson, M. Bishr, T. Stojanovic, J. Ortmann, F. Reitsma, and D. Medyckyj-Scott. eScience for Sea Science: A Semantic Knowledge Infrastructure for Marine Scientists. In *5th IEEE International Conference on e-Science, Oxford, UK, December*, pages 9–11, 2009.

[45] J. Timm and G. Gannod. Grounding and execution of OWL-S based semantic web services. In *Services Computing, 2008. SCC '08. IEEE International Conference on*, volume 2, pages 588–592, July 2008.

[46] P. Vretanos. Web Feature Service Implementation Specification. *OpenGIS project document: OGC*, pages 02–058, 2002.

[47] S. Yau and J. Liu. Service functionality indexing and matching for service-based systems. In *Services Computing, 2008. SCC '08. IEEE International Conference on*, volume 1, pages 461–468, July 2008.

[48] S. S. Yau and J. Liu. Functionality-based service matchmaking for service-oriented architecture. *Autonomous Decentralized Systems, International Symposium on*, 0:147–154, 2007.

[49] S. Yuan, J. Shen, and J. Yan. A practical geographic ontology for spatial web services. In *Services Computing, 2008. SCC '08. IEEE International Conference on*, volume 2, pages 579–580, July 2008.

[50] P. Yue, L. Di, W. Yang, G. Yu, and P. Zhao. Semantics-based automatic composition of geospatial Web service chains. *Computers and Geosciences*, 33(5):649–665, 2007.

[51] R. Zhang, I. Arpinar, and B. Aleman-Meza. Automatic Composition of Semantic Web Services. In *Proc. of the 2003 Int. Conf. on Web Services*, 2003.

[52] D. Zhovtobryukh. A petri net-based approach for automated goal-driven web service composition. *Simulation*, 83(1):33, 2007.

Vita

The author graudated from the University of Chicago in 2003 with a B.S. with Honors in Mathematics, a B.S. in Computer Science, and a B.A. in Near Eastern Languages and Civilizations. He received a M.S. in Computer Science from the University of New Orleans in 2006. He has been working in the Geospatial Sciences and Technology branch of the Naval Research Laboratory at Stennis Space Center since 2004.