Utah State University
DigitalCommons@USU

All Graduate Theses and Dissertations | Graduate Studies

8-2013

Functional Ontologies and Their Application to Hydrologic Modeling: Development of an Integrated Semantic and Procedural Knowledge Model and Reasoning Engine

Aaron Range Byrd
Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/etd

Part of the Hydraulic Engineering Commons

Recommended Citation

Byrd, Aaron Range, "Functional Ontologies and Their Application to Hydrologic Modeling: Development of an Integrated Semantic and Procedural Knowledge Model and Reasoning Engine" (2013). *All Graduate Theses and Dissertations*. 1739.
https://digitalcommons.usu.edu/etd/1739

This Dissertation is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Theses and Dissertations by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.

FUNCTIONAL ONTOLOGIES AND THEIR APPLICATION TO HYDROLOGIC MODELING:

DEVELOPMENT OF AN INTEGRATED SEMANTIC AND PROCEDURAL

KNOWLEDGE MODEL AND REASONING ENGINE

by

Aaron Range Byrd

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

DOCTOR OF PHILOSOPHY

in

Civil and Environmental Engineering

Approved:

Dr. David G. Tarboton Major Professor	Dr. Daniel Watson Committee Member
Dr. Luis Bastidas Committee Member	Dr. Fred Ogden Committee Member
Dr. David Rosenberg Committee Member	Dr. Mark R. McLellan Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2013

This work is produced under funding from the

U.S. Government and is free from copyright

ABSTRACT

Functional Ontologies and Their Application to Hydrologic Modeling: Development of an Integrated Semantic and Procedural Knowledge Model and Reasoning Engine

by

Aaron Range Byrd, Doctor of Philosophy

Utah State University, 2013

Major Professor: Dr. David G. Tarboton
Department: Civil and Environmental Engineering

This dissertation represents the research and development of new concepts and techniques for modeling the *knowledge* about the many concepts we as hydrologists must understand such that we can execute models that operate in terms of conceptual abstractions and have those abstractions translate to the data, tools, and models we use every day. This hydrologic knowledge includes conceptual (i.e. semantic) knowledge, such as the hydrologic cycle concepts and relationships, as well as functional (i.e. procedural) knowledge, such as how to compute the area of a watershed polygon, average basin slope or topographic wetness index.

This dissertation is presented as three papers and a reference manual for the software created. Because hydrologic knowledge includes both semantic aspects as well as procedural aspects, we have developed, in the first paper, a new form of reasoning engine and knowledge base that extends the general-purpose analysis and problem-solving capability of reasoning engines by incorporating procedural knowledge, represented as

computer source code, into the knowledge base. The reasoning engine is able to compile the code and then, if need be, execute the procedural code as part of a query. The potential advantage to this approach is that it simplifies the description of procedural knowledge in a form that can be readily utilized by the reasoning engine to answer a query. Further, since the form of representation of the procedural knowledge is source code, the procedural knowledge has the full capabilities of the underlying language. We use the term "functional ontology" to refer to the new semantic and procedural knowledge models. The first paper applies the new knowledge model to describing and analyzing polygons.

The second and third papers address the application of the new functional ontology reasoning engine and knowledge model to hydrologic applications. The second paper models concepts and procedures, including running external software, related to watershed delineation. The third paper models a project scenario that includes integrating several models. A key advance demonstrated in this paper is the use of functional ontologies to apply metamodeling concepts in a manner that both abstracts and fully utilizes computational models and data sets as part of the project modeling process.

(189 pages)

PUBLIC ABSTRACT

Functional Ontologies and Their Application to Hydrologic Modeling: Development of an Integrated Semantic and Procedural Knowledge Model and Reasoning Engine

by

Aaron Range Byrd, Doctor of Philosophy

Utah State University, 2013

Major Professor: Dr. David G. Tarboton
Department: Civil and Environmental Engineering

In hydrology we straddle the domains of science and engineering. As hydrologists our goal is to predict the movement and volume of water. As scientists we seek to improve our understanding of water-related processes and how to model them. As engineers we seek to be able answer specific water-related questions to provide protection and an essential resource for the people we serve. Underlying all of our work is a body of knowledge that we have developed and continue to develop. This knowledge involves many aspects, such as the role of various hydrologic processes, how to obtain data, computational models that have been developed, and many other things. Knowing this body of knowledge is the key to being a hydrologist. The goal of this work is to enable a computer to begin to think as we do, to reason over hydrologic processes, to deduce what tasks need to be accomplished to answer the hydrologic questions we are asked. To do that we must be able to model how we think as hydrologists, to capture the concepts and procedures we use in a form that a computer can understand.

This dissertation creates a reasoning engine that is able to include both semantic (concept) knowledge as well as procedural knowledge. This new form of knowledge model is called a “functional ontology.” To demonstrate the utility and power of the reasoning engine several functional ontologies are created that capture knowledge about delineating watersheds, knowing how to set up and run computational models, as well as how to create a chain of models to answer the “what-if” questions we are asked. The work shown in this dissertation demonstrates how, through a reasoning engine that combines semantic and procedural knowledge, we can actually model many of the concepts and simple tasks we do as hydrologists in a form that enables the reasoning engine to use deductive logic and automate many of the tasks we do as hydrologists. The focus of this work has been to enhance the use of the tools we as hydrologist use now to examine and engineer solutions to hydrologic problems. We hope that in the future the reasoning tools and knowledge models are further developed to enable a wide range of automated watershed analysis and model creation processes.

This work is dedicated to my wife, Marisol, without whom I would not have accomplished this work; my children, Adán, Mateo, Jacob, and Julia, may they be inspired in their educational goals; my mother, Carol, who always supported and encouraged me; and my father, Harold, who set the example for me and encouraged me to further my educational goals.

ACKNOWLEDGMENTS

I want to thank my advisor, David Tarboton, who persevered and helped me through many drafts and re-writes, always supporting and encouraging me. He also had many helpful suggestions, both for directing the research and in how to more effectively present the work. I want to thank Jeff Holland, the director of the Engineer Research and Development Center, for funding the research program and long-term training opportunity that paid for my time on this project. I also want to thank Jim Westervelt, Jinelle Sperry, and Aaron Lee who created the computational models used in Chapter 4. I also want to thank Sheri Wallace for the many edits to the papers and chapters; my committee for having the patience and faith in me to let me work on a topic that is quite different from typical hydrology dissertations; Rob Wallace for supporting and encouraging me while out in Vicksburg, MS; Earl Edris for allowing me to hide out and escape other work assignments so that I could focus on my dissertation; and David Wilcox for letting me use his office to avoid my office and get a lot of the writing done. I will be forever grateful for my wife, Marisol, who has patiently suffered through this process, picking up and moving our family around to go to school in Logan, Utah and then return to Vicksburg, Mississippi, taking up my slack at home while I was writing, and being absolutely supportive of me the whole way through this process.

Aaron Byrd

CONTENTS

Page

ABSTRACT iii

PUBLIC ABSTRACT v

ACKNOWLEDGMENTS viii

LIST OF TABLES xiv

LIST OF FIGURES xvi

CHAPTER

1 INTRODUCTION 1

1.1 Research Objectives and Overview 2
1.2 Background Concepts 7

1.2.1 Data as Declarative Information 8
1.2.2 Declarative Information vs. Procedural Information 9
1.2.3 Concept-Structured Data 10
1.2.4 Reasoning Logic and Knowledge Models 13
1.2.5 Terminology 16
1.2.6 A Brief Primer on Semantic Modeling 17

1.2.6.1 The Semantic Web World-view 18
1.2.6.2 Creating Concepts 18
1.2.6.3 Community Logic Standards 20
1.2.6.4 Reasoning Engines and Rules 22
1.2.6.5 Including Procedural Knowledge 23

1.3 Related Fields: Model Driven Engineering 24
1.4 Related Practical Applications 28
1.5 Conclusions and Direction 30
References 31

2 ENABLING APPLIED GEOPHYSICAL CONCEPTUAL AND FUNCTIONAL KNOWLEDGE MODELING: INCORPORATING PROCEDURAL KNOWLEDGE INTO A SEMANTIC REASONING ENGINE 37

Abstract 37
2.1 Introduction 38

2.1.1 Semantic Networks and Reasoning........41
2.1.2 XML and Semantic Modeling........42

2.2 Methods........43

2.2.1 API Design........43
2.2.2 Functional Ontology Language Definition........45
2.2.3 Reasonging Algorithm........48
2.2.4 Example........49

2.3 Results........51
2.4 Discussion........52
2.5 Conclusions........57
References........57

3 ENHANCING HYDROLOGIC DATA CREATION AND PROJECT ANALYSES THROUGH PROCEDURAL AND SEMANTIC MODELING........60

Abstract........60
3.1 Introduction........61
3.2 Methods........65

3.2.1 TauDEM Example: Semantic Knowledge Base........65
3.2.2 TauDEM Example: Procedural Knowledge Base........70
3.2.3 TauDEM Example: Project Purpose Knowledge Base........74

3.3 Results........78
3.4 Discussion........79
3.5 Conclusions........87
References........87

4 FACILITATING COMPUTATIONAL MODEL INTEGRATION WITH SEMANTIC AND PROCEDURAL KNOWLEDGE MODELS........90

Abstract........90
4.1 Introduction........91

4.1.1 Background........94
4.1.2 Functional Ontologies........95
4.1.3 Metamodels........95

4.2 Methods........96

4.2.1 Semantic Metamodels of Computational Models........97

4.2.2 Semantic Metamodeling and Procedural Knowledge........................99
4.2.3 Semantic Models of Data Sets 102
4.2.4 Deductive Logic for Data Sets 104
4.2.5 Procedural Modeling for Data Values 107
4.2.6 Modular Ontological Engineering 108
4.2.7 Pulling it all together: Determing Model Integration Requirements 108

4.3 Demonstration Model Integration Project...114

4.3.1 Modeling Domain...116
4.3.2 WRF Scenario Input Data 116
4.3.3 Stochastic Weather Modeling..117
4.3.4 Hydrology and Flatwoods Salamander Modeling 120
4.3.5 Project Specification...124

4.4 Results 124
4.5 Discussion of Results 127
4.6 Conclusions..128
References..129

5 FUNCTIONAL ONTOLOGY CLASS METHODS AND USAGE 131

5.1 Functional Ontology Coding Framework..131

5.1.1 Key Concepts..132
5.1.2 Integrating Code Data into the API 133
5.1.3 Naming Convention..134
5.1.4 Section Overview 135

5.2 Functional Ontology Code Data and Language Reference 135

5.2.1 fo:PrimaryCode 136
5.2.2 fo:SecondaryCode 137
5.2.3 fo:CommonClass 139
5.2.4 fo:UserCode 139
5.2.5 fo:UsingFile...139
5.2.6 fo:UsingNamespace 140

5.3 Reasoning Engine API and User Interface Integration 142
5.4 API Classes 144

5.4.1 BaseOntNode Member and Method Description............................145
5.4.2 NamedNodeSet Member and Method Description.........................145

5.4.2.1 NamedNodeSet::set 147
5.4.2.2 bool NamedNodeSet::SetContains(string setname, string nodename) 147
5.4.2.3 int NamedNodeSet::Union(string list1, string list2, string result) 147
5.4.2.4 int NamedNodeSet::Intersection(string list1, string list2, string result) 147
5.4.2.5 int NamedNodeSet::Difference(string BigSet, string SubtractSet, string result) 148
5.4.2.6 int NamedNodeSet::Size(string name) 148
5.4.2.7 void NamedNodeSet::Merge(NamedNodeSet other) 148
5.4.2.8 string NamedNodeSet::Remove(string name) 148
5.4.2.9 string NamedNodeSet::First(string name) 148
5.4.2.10 bool NamedNodeSet::ExistsSet(string name) 148
5.4.2.11 bool NamedNodeSet::ExistsNonEmptySet(string name) 148
5.4.2.12 void NamedNodeSet::AddToSet(string setname, string concept, FuncOnt the Ontology) 149
5.4.2.13 void NamedNodeSet::AddListToSet(LinkedList<BaseOntNode> list, string name) 149

5.4.3 FuncOnt Member and Method Description 149

5.4.3.1 FuncOnt Constructor 150
5.4.3.2 NamedNodeSet FuncOnt::MakeTempNamedNodeSet() 150
5.4.3.3 LinkedList<BaseOntNode> FuncOnt::MakeBaseOntNodeList() 150
5.4.3.4 string FuncOnt::EncodeLiteral(string value) 150
5.4.3.5 string FuncOnt::DecodeLiteral(string encodedvalue) 151
5.4.3.6 bool FuncOnt::AlreadyInList(string ID) 151
5.4.3.7 AddTriple Methods 151
5.4.3.8 FindMatchingSet Methods 151

5.4.3.8.1 int FuncOnt::FindStandardMatchingSet(string Sub, string Pred, string Obj, NamedNodeSet results) 152
5.4.3.8.2 int FuncOnt::FindMatchingSet(string packedQuery, NamedNodeSet results) 152
5.4.3.8.3 int FuncOnt::FindMatchingSet(string Sub, string Pred, string Obj, NamedNodeSet results) 152
5.4.3.8.4 int FuncOnt::FindMatchingSet(LinkedList<BaseOntNode> Sub, string Pred, string Obj, NamedNodeSet results) 152

5.4.3.8.5 int FuncOnt::FindMatchingSet(string Sub, LinkedList<BaseOntNode> Pred, string Obj, NamedNodeSet results) 153
5.4.3.8.6 int FuncOnt::FindMatchingSet(string Sub, string Pred, LinkedList<BaseOntNode> Obj, NamedNodeSet results) 153

5.4.3.9 bool FuncOnt::ReadRdfXml(string filename) 153
5.4.3.10 void FuncOnt::ReadFromOntologyStore(OntologyStore.OntologyStore theStore) 153
5.4.3.11 LinkedList<string> FuncOnt::GetOntologyNameList() 153
5.4.3.12 bool FuncOnt::WriteOntologyFile(string ontologyName) 154
5.4.3.14 bool FuncOnt::RecompileAll() 154
5.4.3.15 bool FuncOnt::RunUserFunction(string funcName, NamedNodeSet results) 154

5.5 Summary 154
References 155

6 SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS 156

6.1 Summary and Conclusions 156
6.2 Recommendations 163
References 168

CURRICULUM VITAE 170

LIST OF TABLES

Table Page

1-1 Example psuedo knowledge model about hydrologic processes. 14

1-2 Example namespaces and namespace URLs used in the research. 20

1-3 Description of the namespaces referenced in Table 1-2 20

1-4 Example list of RDF, RDFS, and OWL key words used to define namespaces 23

2-1 Namespaces and Namespace URLs used in this research 44

2-2 Description of the namespaces referenced in Table 2-1. 44

2-3 Key words for associating code with concepts in a functional ontology 47

2-4 Test queries for the example ontology 54

2-5 Triples added to the knowledge base as a result of the code execution via the queries 55

2-6 Named node sets as a result of the queries. The set members make true statements from the original query 56

3-1 Code concepts included in the knowledge base 77

3-2 Semantic triples created by the execution of the td: CreateNewTerrainGroup user function 84

4-1 Computational model metamodel predicate definitions 102

4-2 Descriptions of the application-level ontologies developed for this project 110

4-3 Description of the metamodels created for the project 111

4-4 Input and output data sets for the full model integration project 117

4-5 Scale triples for the WRF data used as the input scenario data 119

4-6 Scale triples for the output data sets defined as part of the Savannah Airport stochastic weather model.. 120

4-7 Scale triples for the input data sets for the Savannah Airport stochastic weather model 121

4-8 The scale triples for the input data sets of the salamander model. 123

4-9 The scale triples for the output data set of the salamander model 123

4-10 Initialization triple and reification statements used to denote a particular file to be read in as the WRF historic data 124

4-11 Salamander population results of the two climate scenarios for 10 total runs each 125

LIST OF FIGURES

Figure Page

1-1 Graphical representation of data and data-derived products on the USGS Water Watch web page. Data and imagery from 22 February 2013, http://waterwatch.usgs.gov/. 8

1-2 Schematic of the data or database query information processing paradigm 11

1-3 Reasoning engine query information processing paradigm 16

1-4 Hierarchy of metamodeling layers. 30

1-5 Differences in the current paradigm and the functional ontology paradigm for using semantic models with source code. 31

2-1 Concept map for a simple knowledge base about polygons 50

2-2 Primary code definition for the poly:hasArea predicate 51

2-3 Common class definition for the poly:GetPoints concept 53

2-4 Primary code definition for the poly:hasPerimeter predicate 54

2-5 Point strings for the test polygons 54

2-6 The reasoning engine GUI running the functional ontology for the polygon example 55

2-7 The reasoning engine GUI after the results of the query 57

3-1 The TauDEM triples relating to input and output data sets for the pit remove and 8-way flow direction programs 63

3-2 Part of the TauDEM knowledge base that utilizes subclasses and sub-properties 71

3-3 The concept map for the terrain data group 72

3-4 Example instance of a terrain data group 72

3-5 Semantic linkages between the previous concept graphs and some of the code definitions 76

3-6 The analysis context (project purpose) 80

3-7 Example of an additional analysis context 81

3-8 Contours of the raw DEM file used inthe demonstration example 82

3-9 Watershed basins produced by the TauDEM suite of tools, the TauDEM knowledge base, and the functional ontology API 83

3-10 Excerpt from the log produced by running the td:CreateNewTerrrainDataGroup user method and choosing the single-basin project purpose 85

4-1 Simple semantic model and metamodel of computational models 100

4-2 "Meta" conceptual model of computational models 101

4-3 Semantic model of data sets includes several attributes that act as dimensions to the data set 105

4-4 Representation of the relationships between data, scale, and scenario for a data set 106

4-5 An ontology of scale 109

4-6 Relationship between the several ontologies used to create the semantic and procedural models of conceptual models 112

4-7 A simple example showing a set of models, existing data sets, project descriptions, and the final workflows deduced and executed 114

4-8 Locations of the ecological model domain and the Savannah/Hilton Head Airport 118

4-9 Precipitation statistics used for the two climate scenarios 119

4-10 Temperature statistics for the two climate scenarios 119

4-11 Photos of ephemeral ponds at Fort Stewart 122

4-12 The Flatwoods Salamander model at initialization in the NetLogo environment 125

4-13 Part of the log output by the procedural knowledge of the several ontologies created for this work 129

5-1 Data code for the poly:hasArea predicate function 136

5-2 Method header for the predicate procedure 136

5-3 Complete code wrapping the 'poly:hasArea' predicate method 138

5-4 RDF/XML data for secondary code 138

5-5 Wrapped code for the secondary code 'DoesTerrainGroupHaveActualData' of the primary predicate 'td:hasComputableDependents' 141

5-6 Common code example RDF/XML example 142

5-7 Wrapped common code from Figure 5-6 143

5-8 RDF/XML user code for an 'Add Outlet' procedure 144

5-9 The wrapped user code shown in Figure 5-8 144

5-10 Possible interaction mechanisms between the reasoning engine class and a user interface 146

6-1 The creation of the digital perceptual model from a generalized set of hydrologic model concepts 166

6-2 The creation of the digital conceptual model from the digital perceptual model 167

6-3 The creation of the numerical model input data from the digital conceptual model and data 167

CHAPTER 1

INTRODUCTION

The modeling of water resource phenomena is currently undergoing a significant transformation. Several intersecting factors are driving this change. First, access to public water resource databases using web-enabled tools has made vast quantities of data available in near real-time (Piasecki et al., 2010; NWIS, 2013). Second, computational tools, including faster multi-core hardware and better, physics-based algorithms have drastically reduced the time necessary to compute extremely complex multi-dimensional solutions covering extremely large physical domains (Downer and Ogden, 2004; Kollet and Maxwell, 2006; Qu and Duffy, 2007). Third, there has been a drive to integrate; either through model coupling or algorithmic inclusion; environmental, ecological and atmospheric and other physical processes to create software that better simulates the interconnected physical processes that drive real world decisions (Merritt et al., 2009; Lawrence et al., 2011). Fourth, there has been a significant amount of both research and application of hydrologic processes, such that many processes and their applicability are well understood. These factors are driving model developers to create modeling tools that are vastly more effective at representing the real world. The increase in effective real-world representation spurs the acceptance of the models which in turn pushes the modelers towards the creation of models with larger and larger extent.

Despite the many data collection and computational advances that have occurred, creating large-extent, high-resolution, multi-faceted hydrologic models can be prohibitively expensive, time consuming, and error-prone, especially in light of the trend towards larger and increasingly complex models. This is compounded by challenges manipulating the

complex data required to simulate a complex water resource problem. In spite of our modeling capabilities, concern about project time-to-completion is often just as important as concerns of model appropriateness. In hydrology, just as in most other fields, experience with model development, along with experience in identifying key hydrologic processes in the field, generally results in better models. The hydrologist must integrate knowledge about hydrologic processes, the many techniques of modeling those processes, an understanding of the increasingly many publicly available data sources (and their accuracy), what and how to obtain required data that does not exist (e.g. how to go out and survey data), the implications of the project purpose and project timeline on the quality of the model, techniques for formatting, transforming, and extracting information from data, as well as the particular foibles of the chosen numerical model. The advances in data collection and computational abilities, while yielding advances in our overall capability to study increasingly complex situations, has multiplied the amount of information hydrologists must be able to integrate in order to successfully tackle the larger, more complex problems.

1.1 Research Objectives and Overview

The goal of this dissertation is to research and develop new concepts and techniques for modeling the *knowledge* about the many concepts we as hydrologists must understand such that we can execute models that operate in terms of conceptual abstractions and have those abstractions translate to the data, tools, and models we use every day. This hydrologic knowledge includes conceptual (i.e. semantic) knowledge, such as the hydrologic cycle concepts and relationships, as well as functional (i.e. procedural) knowledge, such as how to compute the area of a watershed polygon, average basin slope or topographic wetness index. Further, I want to develop technologies that allow for the hydrologic knowledge, both

semantic and procedural aspects, to be represented together in a form that enables integrated semantic reasoning and procedural analysis. For example, the reasoning about which hydrologic concepts are applicable to a project could include a procedural analysis of the topographic wetness index. The ultimate, long-term, goal is to be able to represent hydrologic knowledge about processes, implications of project purposes, methods of data investigation, as well as mathematical and numerical models, in order to enable a computer program to reason, using logical and procedural operations on this knowledge, through the process of creating a hydrologic model that is sufficient to answer the questions posed of the system.

Currently, the use of computers to assist in hydrologic analysis is generally confined to the sphere of functional analysis, data conversion, and numerical modeling. Computerized reasoning, on the other hand, over hydrologic concepts is not generally used to enable efficient hydrologic analysis. The primary problem I wish to address is how to use computerized reasoning along with functional analysis to enable an increased range of automated hydrologic analyses.

The specific hypothesis of this research is that knowledge modeling is able to deduce required models and data inputs and facilitate computational model integration in order to compute a desired result based on a specified objective. To test this hypothesis, five sub-hypothesis will need to be demonstrated to be true:

1. Knowledge models can represent and use the forms of knowledge we (hydrologists) use when doing computational modeling;
2. Knowledge models are able to describe and execute computational models;

3. Knowledge models are able to take into account the setting or project purpose when setting up the model execution;
4. Knowledge models are able to represent and integrate a range of computational models
5. Knowledge models are able to deduce which computational models are needful to compute a desired data set

The first four chapters of this work address these five sub-hypotheses. This chapter, Chapter 1, reviews some of the background work and concepts from the field of artificial intelligence. Chapters 1 and 2 discuss current capabilities in the field of knowledge modeling and conclude that, while there are many knowledge modeling tools that have many required knowledge representation aspects, there is no reasoning engine and knowledge model that is able to include the range of knowledge forms used by hydrologists. The specific shortcoming is that there is no mechanism to relate the meaning of procedural knowledge in a manner usable by the reasoning engine.

In order to address this shortcoming, and create a knowledge model that can fulfill the requirements of sub-hypothesis 1, a new knowledge model and reasoning engine is created. Chapter 2 is a paper that presents a new method to integrate semantic and procedural knowledge into a single formal knowledge base. We refer to this as a “Functional Ontology.” The chapter also introduces the reasoning engine developed to operate on this new knowledge base. The reasoning engine, meant to be a proof-of-concept, couples some of the deductive logic capabilities used in other reasoning engines along with an ability to compile and execute procedural knowledge in the form of source code.

Chapters 3 and 4 are papers that demonstrate progressively more complex use cases of the semantic and procedural reasoning engine and knowledge base. Chapter 3 demonstrates how the semantic and procedural knowledge can be used together to model knowledge about watershed delineation. A semantic knowledge base is created to represent the knowledge required to define inputs and execute in the correct sequence tools in the TauDEM watershed delineation toolset, to obtain results appropriate for the given contextual setting and available inputs. This is combined with a procedural knowledge base about creating and running command-line functions which is applied to the execution of TauDEM tools. The knowledge base also includes functionality for transient knowledge and uses it to customize the command line parameters for a set of simple project purposes.

Chapters 2 and 3 together demonstrate that the new functional ontology knowledge model is able to 1) represent and use both procedural and semantic knowledge, the two fundamental forms of knowledge we use as hydrologists, 2) describe and execute computational models, and 3) take into account the purpose of the project when deducing new knowledge.

Chapter 4 details a more complex use case for the reasoning engine, the creation of a logical infrastructure of multiple computational models and data sets, which are referred to as metamodels (De Virgilio, 2010). The purpose of the set of functional ontologies developed for chapter four is to enable computational model integration. The model integration is a loosely coupled paradigm where data is passed from one model to another in sequence and at the end of each model run. The model integration functional ontology is centered around developing one data set from another. Computational models are viewed as the means of transforming one data set into another. Data sets are considered to be windows into actual

data and have several identifying dimensions, including space and time scales, scenario (historic or project alternative), and data type. The data set definition is built on several modular functional ontologies and allows the reasoning engine to deduce which data sets can be created from others, taking into account the many data set dimensions.

The functional ontologies in Chapter 4 create the logical infrastructure for model integration by bringing together the concepts of metamodeling and semantic modeling. The logical infrastructure consists of 1) a metamodel of computational models, 2) a metamodel of data sets, and 3) a metamodel of a typical engineering hydrology project. The logical infrastructure abstracts the data creation process by generalizing the properties of computational models and data sets to create a uniform method of viewing and operating on them. Combining the metamodeling logical infrastructure approach with semantic and procedural knowledge representations of computational models enables the reasoning engine to also be a powerful deductive workflow engine.

The data-centric view of Chapter 4 is in line with decision making processes. Fundamentally, decisions are made based on information we have. Computational models are merely a means to illuminate the implications of data. Because decisions are based on information and the meaning of that information, semantic and procedural models of that information are a viable means to generally inform decisions, rather than just run an analysis. The metamodeling approach for computational models enables the computational models to be a link in the deductive analysis chain for decision making. In the end, it is the decision that changes lives; all of the deductive and analytical powers of the reasoning engine and functional ontologies should be aimed at providing knowledge to inform the decision making process. The demonstration of Chapter 4 is really about creating a process for decision

making that is more than just letting a user pick a model and quickly run it (such as the ARIES models described below), but rather allowing the user to identify the decision data they would like to know about and letting the reasoning engine sort through the process of picking the right set of models to transform existing knowledge into the data used for decisions.

Chapter 4 addresses the fourth and fifth sub-hypotheses, specifically demonstrating a concrete and complex example of when functional ontologies are able to represent and integrate a range of computational models as well as deduce which computational models are needed to compute a desired data set, including taking into account data sufficiency. Together Chapters 1 through 4 demonstrate that functional ontology knowledge models are able to deduce required computational models and facilitate integration between computational models in order to compute a desired result as part of a project.

The fifth chapter is a reference manual for the reasoning engine. It describes the reasoning engine deductive logic process and procedural execution methods. It also specifies the several functional ontology reasoning engine and answer set class methods that can be called by the procedural code in the functional ontology. The final chapter summarizes and discusses the results of the research.

1.2 Background Concepts

The study of how to represent, store, and reason over information in an automated manner has long been the scope of the field of artificial intelligence (Turing, 1950). In hydrology we frequently deal with large amounts of “structured” information, or data. Structured data is data that has a large-scale internal form that is consistent and repetitive.

The other end of the data spectrum is unstructured data, for example the text of a book. It contains a lot of information but not in any regular, pre-defined format.

1.2.1 Data as Declarative Information

Structuring data allows the information to be neatly organized such that it is amenable for automated querying and processing. This data represents specific knowledge about, for example, stream flow or land surface elevation and is termed declarative information. Declarative information is information “about” something and takes the form of statements, but not necessarily in the strictest sense. Figure 1-1, for example, illustrates the output from the USGS’s Water Watch data processing algorithms that work on the volumes of stored stream flow data that the USGS measures. The data are “about” stream stage and flow levels.

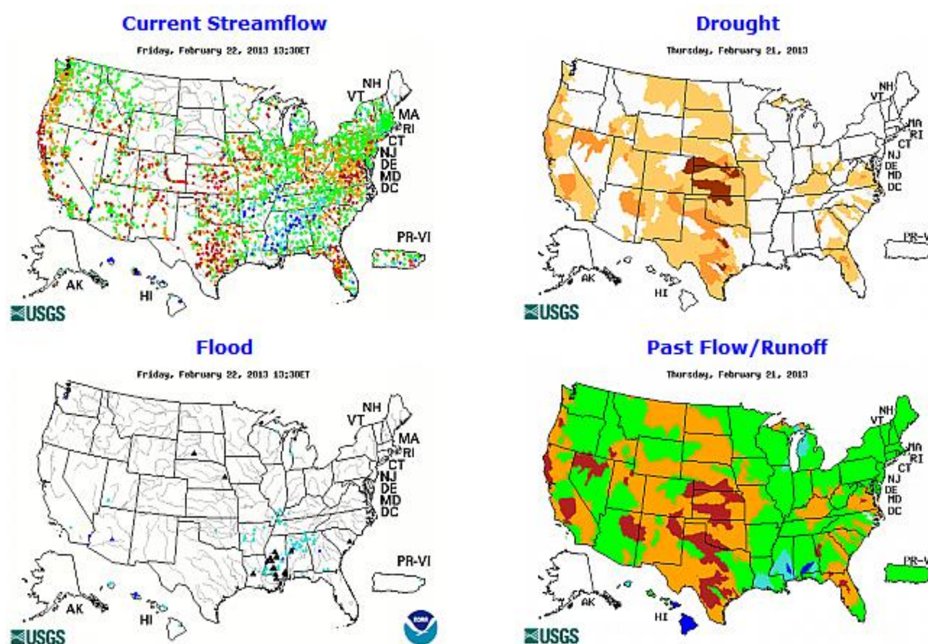


Figure 1-1. Graphical representation of data and data-derived products on the USGS Water Watch web page. Data and imagery from 22 February 2013, http://waterwatch.usgs.gov/.

This approach of relying on a data store (or, more formally, a database) that is queried and interpreted by code that then produces a response is illustrated in Figure 1-1. An advantage of structured data is that, because its form is repetitive, the information about the data, the metadata, can be described separately (and typically only once) from each data item. Systems that use the data in their products are typically built around providing procedures to display and interpret the data (the context, from the metadata sources). This represents the value added by the system. For example, Figure 1-1 illustrates how the USGS, drawing on the same data, can analyze the data using its water watch system to produce graphical depictions of streamflow (and a relative scale of the stream depth), drought indicators, flood indicators, and a comparison of recent data to historical data. The code that displays, processes, and interprets the data can be termed Procedural Knowledge. This is knowledge that encodes the fundamental steps about processing the data.

1.2.2 Declarative Information vs. Procedural Information

The key concept I wish to point out is that data processing and interpretation may be regarded as distinct from the database. All processing and analysis occurs separately from the data. The procedural information represented by a program (or rather the source code for the program) is where information about how to interpret and analyze the data is stored. This procedural information is quite critical to the usefulness of the declarative knowledge.

As a simple, somewhat mundane, example let's consider a program that reads Digital Elevation Model (DEM) data. It could read 1001 DEM file formats and even download DEM data from the web but if someone was to invent a new DEM format (say for enhanced parallel processing of large files) then the program would not have the procedural information it needed to use the data. As another example, say someone invented an

algorithm that calculated a new value that is not part of the existing program, such as the parts of the DEM visible to an orbiting satellite or the accuracy of Global Positioning Satellite (GPS) data based on the satellite's positions, time of day, and topography. A user might need to use this algorithm but would have no way to directly use it without having someone modify the source code of the software they used for working with DEMs, which may or may not be feasible.

The functional ontology approach developed here provides a logical infrastructure foundation that could incrementally encode procedural and declarative knowledge about the new DEM format or new algorithm in order to integrate them into the reasoning engine driven data analysis. Functional ontologies enable the creation of the underlying logical infrastructure for a new class of software that allows for this type of incremental procedural knowledge improvement.

1.2.3 Concept-structured Data

When we think of structured data we often think of data in a database or some similar template-based format such as shapefiles, Digital Elevation Model data, etc. There is another type of structured data, one that makes explicit the relationship between the declarative meaning of the data and the data itself, rather than rely on an implicit relationship between the data and its declarative meaning from its position in the overall data structure (e.g. what is the declarative meaning of the 313th floating point value in an ASCII grid file and where is it located in the real world?) An example of this data format, which I am calling concept-structured data, is Extensible Markup Language (XML) data.

Data Query Information System Paradigm

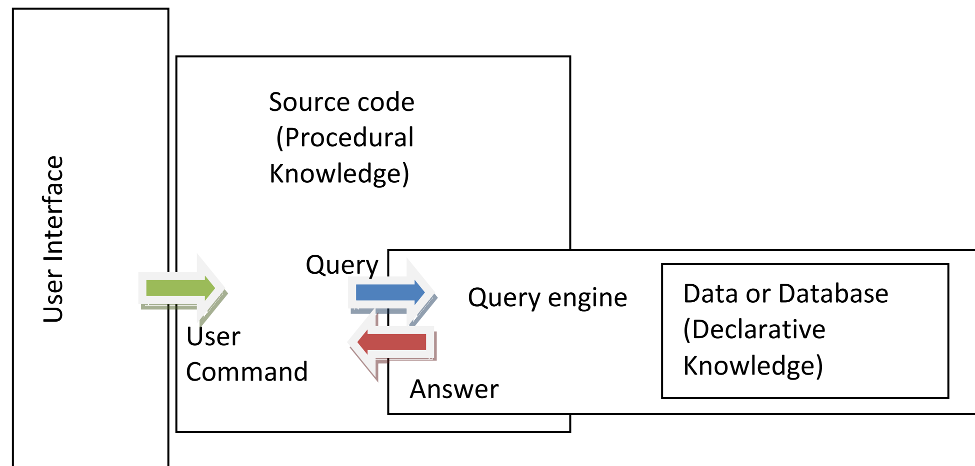


Figure 1-2. Schematic of the data or database query information processing paradigm. The user interacts with a program (either local or remote) that queries the data or database and, once the data is returned, performs value-added processing that interprets and processes the data.

XML can store any data, and has definitions and structure regarding how to describe the information about data (i.e. XML is self-decribing), but is completely open when it comes to what data you want to store, how to name it, and what all can be in the data set. An XML file could just as easily store a stream flow value, a description of a stream bed, a set of points describing contours of a mountain, and your neighbors shoe size, all in one file. The key point here, though, is that the XML tags are used to describe the data – the format facilitates an explicitly defined relationship between the information and its declarative meaning which, in turn, is also “data” in a sense that it is also declratative information. An XML file can hold any type of data, but the data must be enclosed in tags that describe what

it is. These tags are the declarative meaning of the data but are also a form of declarative information themselves (they are data, too, just a different “type” of data – concept data).

To make XML files more amenable to automated processing, XML schemas (W3C) were created to impose structure on the data. These schemas define the types of information that the XML files will hold and also creates a standard set of terminology for the types of information. Examples of these are HyperText Markup Language (HTML) files and the OpenGIS® KML Encoding Standard (OGC). (KML formorly was an acronym for Keyhole Markup Language but now the schema is just called KML.) The structure that these schemas impose on XML documents allows them to be used in an automated fashion by web browsers, in the case of HTML files, and Geographic Information Systems, in the case of KML files. These schemes essentially impose a structure on an XML file and turn it into a structured data set such that Figure 1-3 applies.

An important point, though, is that while the XML schema describes what data goes together and begins to create relationships between one data and another, the interpretation of the meaning of this data is still left to the external program that reads the data. The schema approach also limits what can be in the file in the interest of imposing the advantages of a structure on the concept-based knowledge.

Structurally, the same procedural information limitations apply as before, since it is still separate from the declarative information. For example, a KML file may list the points for a polygon but the procedural information in a GIS can check to see if one polygon overlaps another, intersect the two, and compute the resulting areas of all the polygons. If someone were to invent a better algorithm that could process thousands of polygons in a fraction of the time it takes for the original algorithm to do one polygon, there is no direct way to

incorporate this into the procedural information, or procedural knowledge base, without rewriting portions of the GIS code.

1.2.4 Reasoning Logic and Knowledge Models

Knowledge models are a type of structured data about concepts. Unlike the concept-structured data, the concepts themselves are the data. For example, a data “model” of stream flow could list a set of flow values and their connectivity over time. A knowledge “model” would list a set of concepts and how they are related.

Knowledge models are a form of declarative knowledge, just as data about stream flow is. They, however, also have a logic undergirding them that allows for a reasoning engine to not only perform queries but also perform deductive reasoning. The reasoning engine itself has procedural knowledge encoded in it that can interpret the logic terms rather than simply return values to be interpreted by the calling program. For example, software for working with time series of flow values has the implicit knowledge that the flow values are sequentially connected by time and that they are flow values. Because of this implicit knowledge it can display hydrographs and discover (via a computational algorithm) the total flow volume. This knowledge and the data relationships are utilized by the program interacting with the user, not whatever querying engine or algorithm is used.

A reasoning engine, on the other hand, could, independent of the software interacting with the user, perform analyses that discover data that is not explicitly represented in the knowledge model. Consider the following example (Table 1-1) pseudo knowledge model. Note that the data is organized into sentences with a subject, a predicate, and an object. The subject, predicate (verb), and object are all concepts – declarative information – but the form of the statement creates a relationship between them. We use

Table 1-1. Example pseudo knowledge model about hydrologic processes and storages. The data are sentences in the form of simple statements.

<Subject>	**<Predicate>**	**<Object>**
Hydrologic Process	"is a"	"Class"
Hydrologic Storage	"is a"	"Class"
Moves Water From	"has Domain"	Hydrologic Process
Moves Water From	"has Range"	Hydrologic Storage
Moves Water To	"has Domain"	Hydrologic Process
Moves Water To	"has Range"	Hydrologic Storage
Precipitation	Moves Water From	Atmosphere
Precipitation	Moves Water To	Overland Surface
Infiltration	Moves Water From	Overland Surface
Infiltration	Moves Water To	Vadose Zone
Percolation	Moves Water From	Vadose Zone
Percolation	Moves Water To	Groundwater
Exfiltration	Moves Water From	Groundwater
Exfiltration	Moves Water To	Overland Surface
Subsurface Discharge	Moves Water From	Groundwater
Subsurface Discharge	Moves Water To	Streams
Subsurface Discharge	Moves Water To	Ocean
Stream Flow	Moves Water From	Streams
Stream Flow	Moves Water To	Ocean
Evaporation	Moves Water From	Ocean
Evaporation	Moves Water From	Streams
Evaporation	Moves Water From	Overland Surface
Evaporation	Moves Water From	Vadose Zone
Evaporation	Moves Water To	Atmosphere

this form every day in our speech and thoughts to convey complex information that includes relationships between concepts.

Without going into details at the moment, a reasoning engine operating on this knowledge model could answer questions such as “What are hydrologic processes?” and “What are hydrologic storages?” that are not explicitly stated in the knowledge model as well as questions like “what processes move water to and from groundwater?” that are explicitly stated. The ability of the reasoning engine to interpret and deduce new information is due to

both the logic algorithms built into the reasoning engine and the logic concepts built into the knowledge model, in this case the “has Domain,” “has Range,” “is a,” and “Class” concepts.

The goal of constructing a knowledge model is to enable a computer to glean and deduce information in answer to a query. Reasoning engine Applications Programming Interfaces (APIs) are built to operate on formalized knowledge models and attempt to answer user submitted queries. In essence queries are simple questions asked of the reasoning engine such as <A P ?B> which means “Given a subject **a** and a predicate **p**, what is the set of concepts B (with members $\mathbf{b}_i$) that are true statements <**a p** $\mathbf{b}_i$>.” The reasoning engine uses pattern matching and the logic statements to deduce the set of concepts that match the given query parameters. The result is a set of concepts B = {$\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$.}. Another type of query is a truth-test. In this case the query returns whether or not the query triple has been asserted as a truth.

The reasoning engine model of interaction with an end-user program is similar to the database model. In Figure 1-3 the database from Figure 1-2 has been replaced by a formalized knowledge base (e.g. OWL) and query engine replaced by a reasoning engine. In other respects the paradigms are the same. The strength of the knowledge model and reasoning engine approach to information processing is two-fold: it deals with abstractions of concepts as well as it is able to augment the data in the knowledge base through the reasoning logic algorithm. There are a few shortcomings of knowledge models, though. Knowledge models are excellent at working with abstractions but not effective at applying the meaning of the abstractions outside of the logic terms programmed into the reasoning algorithm. Additionally, there are questions about knowledge model logic forms even being able to adequately describe the breadth of knowledge representations of interest. For

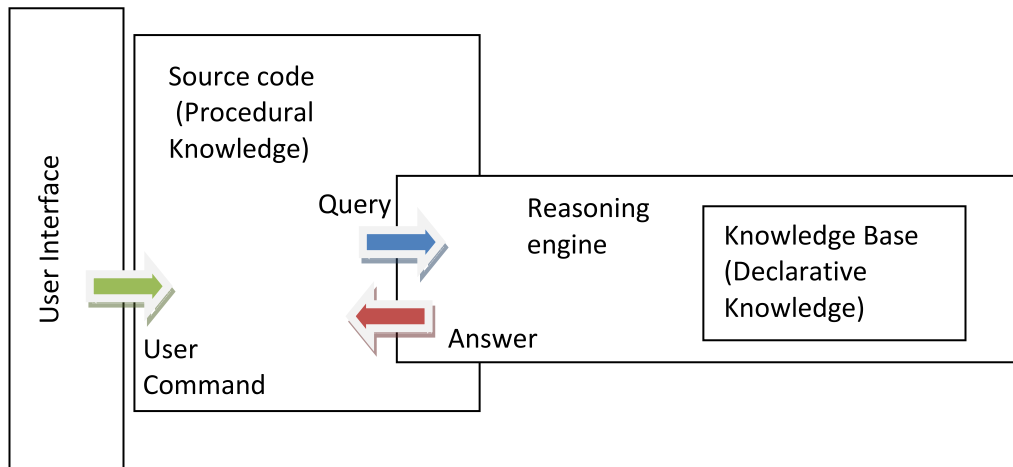


Figure 1-3. Reasoning engine query information processing paradigm. This paradigm is similar to the database information processing paradigm with the primary difference that the reasoning engine can interpret and augment the knowledge base through the reasoning logic algorithm.

example, Freitas and Lins (2012) discuss the limitations of current knowledge models to capture mathematical formulations in ontologies.

1.2.5 Terminology

In *semantic modeling*, knowledge (information) is encoded using a group of concepts linked in a graph or web-like manner to represent relationships between the concepts. *Reasoning engines* are created to operate on these concepts and relationships and logically deduce consequent knowledge from the existing knowledge. In a general sense a group of concepts, whether in a semantic model or some other form, can be termed a *knowledge base* or *knowledge model*. A set of concepts and relationships between the concepts, formalized and expressed as a semantic model, is termed an *ontology*. Ontologies in computer science stem from the philosophical study of “being” or in other words, what

something is. Related to ontology is mereology, or the study of part-whole relationships. In computer science, ontologies are used to represent and store knowledge about objects, things, or general concepts. The fundamental goal of an ontology is to represent and define relationships between concepts. Reasoning engines depend on formalized semantic logic terms, derived from first-order logic, to deduce consequent knowledge. For example, from the statements ("A is a class of things," "All A's have color Red," "B is a type of A") a reasoning engine could deduce that "B has color Red."

1.2.6 A Brief Primer on Semantic Modeling

Semantic modeling focuses on developing models of concepts and the relationships between them. There are several advantages to using semantic models to represent geophysical knowledge: semantic models provide a common language for scientific interoperability of digital products, semantic models can capture knowledge in a framework that allows for automated, reproducible reasoning, and the reasoning logic capabilities can deduce knowledge not readily apparent, especially with cross-discipline knowledge.

The underlying philosophy of semantic modeling is that "meaning" is entirely a function of the relationships between concepts. Semantic modeling utilizes graph theory (which began with Euler, 1741) to represent the relationships between concepts in order to inform automated reasoning tools how to infer additional relationships between concepts. Undergirding the reasoning tools is a logic called descriptive logic (Ceccato, 1961; Masterman, 1961; Brachman, 1979; Sattler et al., 2009) (see Sowa, 1992) which is derived from first order logic (Peirce, 1885; Frege, 1879; Gödel, 1929).

1.2.6.1 The Semantic Web World-view

One of the primary groups of ontology standards and tools in use today comprises the Semantic Web (Berners–Lee et al., 2001). According to Berners-Lee, the Semantic Web is a way of classifying information into ontological descriptions such that ontology-enabled web tools can search and find data related to other data in a meaningful way. The components of the Semantic Web can be described as being similar to layers of a cake (Berners-Lee, 2007). The upper layers, such as the semantic languages, user applications, etc., serve to add meaning and functionality to the lower, foundational layers (such as URL/URIs, XML, RDF.) The lower layers create the ontology framework while the upper layers work to infer information from the ontology and utilize it in some fashion. Ontologies use a knowledge description format that is built on URIs, XML, and RDF. Functionality is created by reasoning engines (RDF-S, OWL) that work through queries (SPARQL) and apply rules (RIF). The unifying logic, proof, trust, and cryptography components are important to the semantic web but external to the workings of the ontology. The User Interface and Applications utilize the ontology to do the designated tasks.

1.2.6.2 Creating Concepts

The foundations of the Semantic Web are Uniform Resource Identifiers (URI) and Uniform Resource Locators (URL). URIs are a method of creating unique identifiers for a resource. If that resource happens to be on the web then the URI created is a URL, such as http://www.w3.org/2000/01/rdf-schema#subClassOf. The URL relates a resource (a text document in this case) to a unique identifier. URIs, though, are very general in nature. A URI for a book, for example, would be the ISBN number.

XML statements function as a mechanism to associate groups of concepts together and to assign names and a simple hierarchy to the members of the group. XML Metadata standards, such as the Dublin Core Metadata Initiative (DCMI, 2012), facilitate reuse of information by defining the meaning of terms. Name spaces in XML are a particularly useful concept in that a whole group of statements or concepts can be associated together and referenced in a remote location without having to duplicate the original information.

The goal of an ontology is to model concepts. Hopefully this is done in a manner that facilitates modularity and re-use of the ontologies (see Grau et al., 2008). One of the key aspects of making an ontology modular is to create a reference set of concepts and to publish those concepts such that they are globally available and unique. Uniform Resource Identifiers (URIs), Uniform Resource Locator (URLs), and XML namespaces are a means of creating and referencing globally unique concepts. The "tag" URL format is used to create globally unique identifiers, such as http://www.example.org/MyOntology.rdf#Concept. Using an XML namespace (ex = http://www.example.org/MyOntology.rdf#) further simplifies the formatting and enhances the human readability of the concept (ex:Concept.) A benefit of using URLs is that there can actually be a web page at the URL that describes the concepts in human terms and as well as host a file to be downloaded and used by a computer. Table 1-2 and Table 1-3 show some example namespaces, including the namespaces used in the examples and test cases for this research. Resource Description Framework (RDF) (Klyne and Carroll, 2004), Resource Description Framework – Schema (RDF-S) (Brickley and Guha, 2004), and Web Ontology Language (OWL) (Patel-Schneider et al., 2004) are description logic languages used in the Semantic Web (Berners–Lee et al., 2001).

Table 1-2. Example namespaces and namespace URLs used in this research.

Namespace Symbol	Namespace URL Reference
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
owl	http://www.w3.org/2002/07/owl#

Table 1-3. Description of the namespaces referenced in Table 1-2.

Namespace Symbol	Description
rdf	Resource Description Framework (RDF) is a description logic standard that include some basic logic and the specification of the triple format for storing logic statements. Also specifies an XML file format for ontologies denoted RDF/XML.
rdfs	Builds upon the RDF specification, Resource Description Framework – Schema (RDF-S) includes additional logic for class / subclass relationships
owl	An advanced description logic standard used throughout the semantic web. OWL stands for Web Ontology Language. Includes class / subclass, inverse, equivalence, restrictions, cardinality, and many other types of logic. Often serialized in RDF/XML format.

Since URIs are used for ontologies, both in keywords for ontological standards and general ontology terms created for a particular application, it is generally encouraged that if the URI is a web page reference then it should resolve to an actual web page with a written description of the concept.

1.2.6.3 Community Logic Standards

The Resource Description Framework (RDF) (Brickley and Guha, 2004) encodes all of the information in the ontology in a subject-predicate-object relationship referred to as a "triple." For the semantic web, the RDF encoding is in XML. One of the rules about the Semantic Web is that "Anyone can say Anything about Any Topic" (Allemang and Hendler, 2008). In RDF this translates to allowing any subject, predicate, and object as part of the triple.

Besides being a triple form for encoding connections between concepts, RDF, along with the Resource Description Framework – Schema (RDF-S), has a set of defined predicates and objects that facilitates inference by RDF reasoners. RDF has a relatively simple lexicon, such as being able to define "is a" relationships. RDF-S simply extends the defined set of key words to include class/subclass relationships, along with domain and range descriptors for predicates. The Web Ontology Language (OWL) (Patel-Schneider et al., 2004) adds to RDF and RDF-S an additional set of specified predicates, objects, and their properties that OWL interpreters will know how to use and to make additional inferences, such as logic properties and restrictions.

There are three flavors of OWL that permit varying amounts of logic. These flavors are meant to ensure decidability of the logic to varying degrees.

Ontologies use namespaces to define groups of related properties. For example, Table 1-4 lists several keywords from three different ontological namespaces, RDF, RDF-S, and OWL. The namespace prefix to the keyword is meant to both identify the keyword as belonging to a particular ontological lexicon as well as giving it a unique identifier. The RDF namespace shorthand, rdf:, takes the place of the full rdf namespace, http://www.w3.org/1999/02/22-rdf-syntax-ns#. Thus, rdf:type is actually http://www.w3.org/1999/02/22-rdf-syntax-ns#type, which is a complete, unique URI.

Some ontologies, such as RDF-S and OWL, include keywords from other namespaces, like RDF, illustrating that ontologies are meant to be extendable (Grau et al., 2008). This is a key feature of ontologies. Because of the namespace feature along with the unique URIs, there are a few technical difficulties in merging different ontologies. Generally the

ontological differences, which are usually naming differences, can be resolved by creating a fairly simple bridging ontology that equates concepts from one ontology to another.

1.2.6.4 Reasoning Engines and Rules

To use an ontology, a reasoner and query engine is needed. A popular query engine for RDF and RDF-S is called SPARQL (Prud'hommeaux and Seaborne, 2008). Others for various OWL flavors include Pellet (Sirin and Parsia, 2004) and SQWRL (O'Conner and Das, 2009).

Ontologies excel at describing data and the relationships between data in a manner that allows for both automated data integration as well as inference of new data. The nature of ontologies allows them to describe rules, such as business rules, that can be used to make conditional decisions. The ontology inference engines, though, do not know how to use the rules. The Semantic Web Rule Language (SWRL) (Horrocks et al., 2004) combines OWL with a rule syntax known as RuleML (Boley et al., 2001) to enable the expression and evaluation of rules. The rules allow for further data integration and inference in situations where conditional statements need to be used. Other coupled ontology and rule systems, often used for automated business logic, include JBoss Drools (Browne, 2009), Jena (Carroll et al., 2004), Jess (Friedman-Hill, 2003), and SweetRules (Grosof, 2004).

Semantic Web Services (SWS) (Payne and Lassila, 2004) are a group of young technologies that offer varying approaches to encapsulating web services in an ontology description. According to (Hebeler et al., 2009) there are three primary approaches under development, Semantic Markup for Web Services (OWL-S) (Martin et al., 2004), Web Service Modeling Ontology (WMSO) (Roman et al., 2005), and Semantic Annotations for WSDL and XML Schema (SAWSDL) (Farrell and Lausen, 2007).

Table 1-4. Example list of RDF, RDFS, and OWL key words used to define meaning in their ontologies. P and Q represent predicates, x, y, and z represent subjects/objects, and the form P(x,y) means that x is related to y via predicate P, or <x> <P> <y> in triple form.

Keyword	Language	Meaning
rdf:Type	RDF	"is a" predicate
rdf:Property	RDF	define a predicate
rdf:Resource	RDF	define a object/subject
rdfs:Class	RDFS	define a group type
rdfs:domain	RDFS	Indicates the subject class of a predicate
rdfs:range	RDFS	Indicates the object class of a predicate
rdfs:SubClassOf	RDFS	creates a sub-group type
owl:TransitiveProperty	OWL	Indicates that a predicate has the transitive property. Specifically if P(x,y) and P(y,z) then P(x,z) is implied. e.g. west of: X is west of Y, Y is west of Z, therefore X is west of Z
owl:SymmetricProperty	OWL	Indicates that a predicate has the symmetry property. Specifically if P(x,y) then P(y,x) is implied, e.g. equality: if A=B, then B=A
owl:InverseOf	OWL	Indicates that a predicate has an inverse predicate. Specifically if Q InverseOf P then P(x,y) implies Q(y,x), e.g. P=less than, Q=greater than: if A<B then B>A
owl:equivalentClass	OWL	Two object/subjects are the same (A is actually the same concept as B)
owl:Restriction	OWL	Narrows the scope of possible attributes (e.g. cardinality: must have 3 attributes)

1.2.6.5 Including Procedural Knowledge

Similarly to the point raised in section 1.2.2, Hartley (1985) argues that procedural knowledge is a separate but equally important knowledge base as semantic knowledge. Encoding and enabling the computer to execute procedural knowledge along with the semantic reasoning is the goal of a class of semantic models termed ‘attached procedure executable semantic networks’ (Sowa, 1992). One class of attached procedure executable semantic networks are semantic models describing actions that an actor may take when

certain conditions are met (Hartley, 1984). The reasoning engine looks to match a set of preconditions for the execution of the procedures and, if found to be true, executes the sequence of steps specified. This sequence of steps is encoded as a set of concepts that the reasoning engine must interpret.

A new technology, Answer Set Programming (Schindlauer, 2008), augments semantic rules (discussed by, e.g., Boley et al., 2001; Horrocks et al., 2004) with the ability to query the knowledge base as part of the rule (Eiter et al., 2008). This creates functionality where the rules are able to be based on the current content of the knowledge base. Semantic rules and answer set programming are designed to augment basic logical constructs of description logic languages with complex second and higher order logics, conditional logic, and logic statements based on the answer to semantic queries. Ontologies (semantic networks based on a description logic language) coupled with rule sets are termed Description Logic Programs (Grosof et al., 2003; Motik and Rosati, 2007).

1.3 Related Fields: Model Driven Engineering

A field related to semantic modeling is that of Model Driven Engineering (MDE). The model engineering paradigm is based on the principle that “everything is a model” (Bézivin, 2005) and has the goal of disciplined and rationalized production of models (Favre, 2005). MDE uses a similar concept to semantic modeling in that models are sets of concepts and relationships between them. In this light, MDE has progressed to specify not only how to abstract information into a model but also how to, in turn, model the models, or in other words, how to create metamodels. Metamodels are a description of the format that actual models should have. There is even a tier higher termed metametamodels or more simply megamodels. This concept is illustrated in Figure 1-4. At the base (M0 layer) are things and

instances (e.g. numerical hydrologic model processes, actual physical processes in a watershed, or web pages describing watershed properties). In metamodeling, the goal is to describe these things and instances in terms of concepts, which is the M1 layer. This layer roughly corresponds to the RDF layer in an ontology.

The M2 layer is metamodels that describe the framework of the M1 layer. In one sense they are like the additional logic properties of the ontologies, such as class relationships and predicate properties. They are also used, however, to structure different ways of representing the same basic concept. For example, an M2 layer could describe the content structure for a set of web pages that have the same content but different visual representations. The M1 layer would be the individual web page layouts. Another example would be that the M1 layer could describe various infiltration models while the M2 layer dictates the how the data for the infiltration models should be represented such that the same set of information could be used for several infiltration models. In other words, the data model for each infiltration model is an M1 model, and how those data models should be framed is an M2 metamodel.

Above the M2 layer is the M3 layer, the Meta-meta model (or mega-model) layer, that describes how the metamodel is stated. In an ontology, this layer translates to the set of overarching concepts, such as the completeness of the description logics used in the reasoning engine (Figure 1-4).

While somewhat similar to ontologies, MDE is not generally used for the same kinds of reasoning tasks as ontologies. One of the primary benefits, though, of this metamodeling approach is that data, information, or knowledge can be transformed from one format to another by constructing a schema translation based on the level above it. In fact, not only

data can be transformed but models can be transformed as well. For example, while MDE is used to model many physical, digital, and conceptual systems, it is also frequently used to model software. Diagrams of variables and related code can be described in a general format and then translated to any number of programming languages. Gaševic et al. (2009) discuss several areas of research in turning MDE diagrams into workable code. Another example of a very pertinent tool that has been developed and used for transformations is XML Schema Language Transformations (XSLT) (Kay, 2007). XSLT can translate from one XML document into another by using an XML schema document as a template. The XML schema document serves as the metamodel for translating one into the other. Model translation is a very active area of research.

A commercial application of this concept has been developed in the software MapForce® by Altova® (http://www.altova.com/mapforce.html.) The software can take an XML data file and convert it into a host of other data formats, including flat files and databases. It claims to even be able to create software that can be run in a stand-alone fashion in order to automate the process.

An important tool in use today for MDE modeling is Unified Modeling Language (UML) (OMG UML, 2010). UML is fairly analogous to OWL for ontologies. In fact, typical UML constructs are graph diagrams that demonstrate relationships between concepts. Since ontologies can be considered models of concepts in the MDE sense, Gaševic et al. (2009) discuss the issue of being able to convert between ontologies and UML graphs. Overall they find that it is possible but there are a few issues. One, interestingly enough, is that two owl

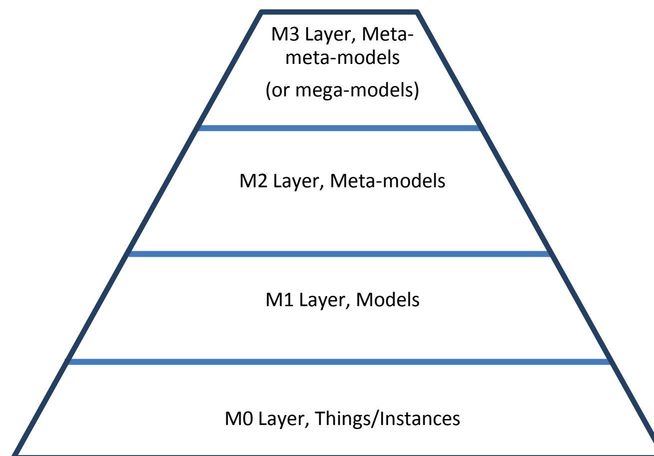


Figure 1-4. Hierarchy of modeling layers. Adapted from (OMG, 2006).

flavors, OWL-DL and OWL-Full, are more expressive than UML and include constructs that cannot be represented in UML.

The modeling layers of MDE and ontologies are related, but not on a one-to-one basis. An ontology where only individuals are presents (e.g. RDF) represents an M1 layer in the MDE scheme whereas an ontology that uses class constructs (e.g. RDFS) as well as individuals would be at both the M1 and M2 layers. Semantic web services (e.g. Martin et al., 2004; Roman et al., 2005) each have their own M2 ontology, a specification for the format of the web service description document. Another important application of metamodels is for data validation (Conejo et al., 2007).

From the description of model driven engineering it is apparent that they are close relatives of the semantic models and many of the foundational concepts are similar. The advantage of MDE is that they are in part designed to represent process descriptions. The downside, though, is that they lack the ability to include deductive analysis and queries as the central function of the knowledge representation.

From this discussion, it is apparent that there is no one single tool able to represent hydrologic knowledge to the degree of facilitating the use of hydrologic tools and computational models. The work done for this dissertation will focus on including procedural models as part of a semantic model and reasoning engine and then demonstrate the application of the semantic and procedural model and reasoning engine for hydrologic situations.

1.4 Related Practical Applications

OntoWEDSS (Ceccaroni et al., 2004) has been developed as a coupled ontology, rule, and sensor system for automating the management of a wastewater treatment plant. It uses an ontology to define the references for measured quantities, perceived system state, and actions that should be taken. The rules translate the measured quantities into the perceived system state as well as prescribe the needed actions.

The OntoWEDSS project is relevant for two reasons. First, that ontologies are used both for integration of data as well as reasoning over necessary implications of the data. Secondly, the framework integrates perceptions from the many sensors available in the waste water treatment plant.

The SEAMLESS project (Janssen et al., 2008) (see also http://www.seamlessassociation.org/) uses an ontology to define the meaning of fields in a relational database that is used by several different models. The goal was to provide a means of agricultural model interoperability through an ontological description of the data used by the disparate models. The data covered many different database fields and came from different data sources. Data included farm activities and income, soils data, climate data, farm management data, and herd levels and output. After several iterations, they were able

to successfully create an ontology that both described the relationships between data fields as well as enabled the creations of individuals that identified specific database fields.

ARIES (Villa, 2009) is a project aimed at using an ontology to describe environmental flux information in order to assist in modeling. By using a standardized ontology describing environmental fluxes, the project goal is to enable models to describe themselves in a fashion that allows for automated model to model linking. Villa describes the processes that occur during a user session:

> During an ARIES user session, users will select an area of interest using an interactive map, and a set of observables (e.g. carbon sequestration or flood protection value) that they want quantified. As soon as the user priorities are set, the ARIES engine will look up semantic models for all the observables of interest and start the two-phase process described above: the context of evaluation of each model will be computed first, and different specialized models for each context state will be built, trained and computed. As the main paradigm for modeling in ARIES is Bayesian network models (Cowell et al., 1999), calibration (training) of the models can be performed in advance and cached in most cases. Users are then able to set forcing functions or change the value of parameters and recompute all models to explore scenarios of interest.[1]

The “observables of interest” are the particular observed values, such as rainfall amount or soil hydraulic conductivity, that are specified by the model. These are stored in a semantically annotated database. The data in ARIES is stored in “knowledge boxes,” or k-boxes. These k-boxes take the form of semantic wrappers around storage formats, such as GIS data or SQL databases. Spatial and temporal scaling was accomplished by using a spatially explicit paradigm in the specification of the context variable with reference to spatial scales from (Wu, 2006).

ARIES, OntoWEDDS, and SEAMLESS all build on semantically-mediated databases. This gives the application a specific, but limited, set of controlled vocabulary to use in the

---

[1] Villa, 2009. Semantically driven meta-modelling: automating model construction in an environmental decision support system for the assessment of ecosystem services flows. Information Technologies in Environmental Engineering, Springer Berlin Heidelberg**:** 23-36.

modeling process. Each of these modeling applications represents a significant amount of work and a significant advance for using semantic modeling with procedural models. The goal of this work is to take the integration of semantic and procedural modeling a step further, to enable general procedural knowledge descriptions to be used as part of the deductive logic process of the reasoning engine.

1.5 Conclusions and Direction

Semantic modeling holds out the promise of significant advances in automating hydrologic processing, especially as evidenced by ARIES, OntoWEDDS, and SEAMLESS projects. Model Driven Engineering, however, appears to be more focused on modeling the models than creating a tool for practical use. It does appear, though, that the advantages of Model Driven Engineering, namely metamodeling, can be gained through the proper application of semantic modeling. These approaches, however, still do not enable a complete treatment of procedural knowledge in order to include the procedural knowledge as an integral part of the reasoning process, such as we use in hydrology. This work will focus on combining procedural knowledge with semantic models and reasoning engines such that the procedural knowledge is able to be used directly by the reasoning engine. The following figure, Figure 1-5, illustrates the differences between the current semantic modeling paradigm and the proposed functional ontology paradigm where the reasoning engine is able to directly execute procedural knowledge. The difference being that the source code about concepts is intimately tied to the concepts within the knowledge base rather than being separate from the knowledge base. This will allow programmers to seamlessly integrate procedures into the knowledge base rather than forcing programmer to implicitly tie the code and semantic concepts together.

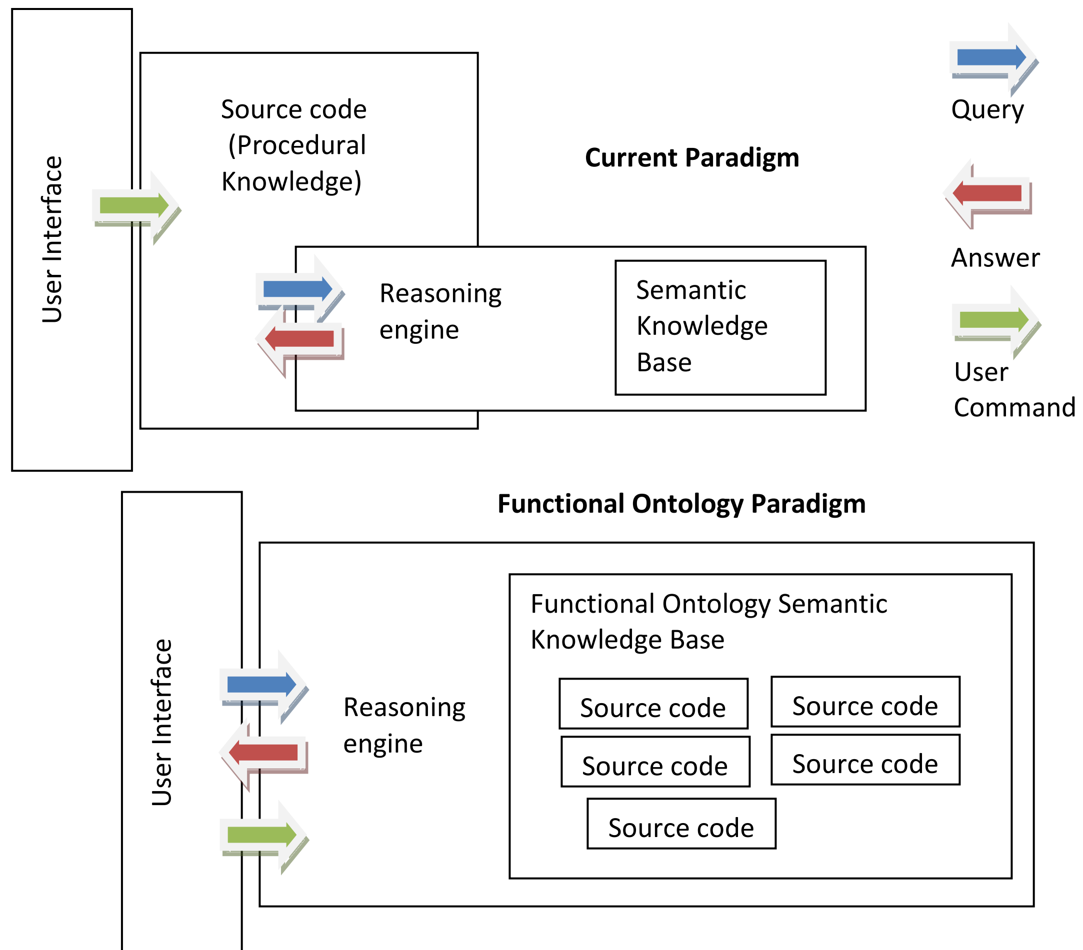


Figure 1-5. Differences in the current paradigm and the functional ontology paradigm for using semantic models with source code.

References

Allemang, D. and Hendler, J. 2008. *Semantic Web for the Working Ontologist.* Burlington, MA, Elsevier.

Berners-Lee, T. 2007. "The Semantic Web Layer Cake Diagram." Retrieved September 29, 2010, from http://www.w3.org/2007/03/layerCake.svg.

Berners–Lee, T., Hendler, J., and Lassila, O. 2001. "The Semantic Web." Scientific American **284**(5):35-43.

Bézivin, J. 2005. "On the unification power of models." Software and Systems Modeling **4**(2): 171-188.

Boley, H., Tabet, S., and Wagner, G. 2001. Design Rationale of RuleML: A Markup Language for Semantic Web Rules. SWWS'01, Stanford.

Brachman, R. J. 1979. On the epistemological status of semantic networks. *Associative Networks: Representation and Use of Knowledge by Computers.* N. V. Findler, ed. New York, Academic Press.

Brickley, D. and Guha, R. V. 2004. RDF vocabulary description language 1.0: RDF schema, W3C.

Browne, P. 2009. JBoss Drools Business Rules, Packtpub.

Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., and Wilkinson, K. 2004. Jena: implementing the semantic web recommendations. 13th international World Wide Web conference. New York**:** 74-83.

Ceccaroni, L., Cortés, U., and Sànchez-Marrè, M. 2004. "OntoWEDSS: augmenting environmental decision-support systems with ontologies." Environmental Modelling & Software **19**(9): 785-797.

Ceccato, S. 1961. *Linguistic Analysis and Programming for Mechanical Translation.* New York, Gordon and Breach.

Conejo, R., Guzmán, E., and Pérez-De-La-Cruz, J.-L. 2007. "Knowledge-based Validation for Hydrological Information Systems." Applied Artificial Intelligence **21**(8): 803-830.

Cowell, R. G., Dawid, P., Lauritzen, S. L., and Spiegelhalter, D. J. 1999. *Probabilistic Networks and Expert Systems.* New York, Springer.

DCMI 2012. Dublin Core Metadata Initiative (DCMI)Metadata Terms. Dublin Core Metadata Initiative Usage Board. http://dublincore.org/documents/2012/06/14/dcmi-terms/

De Virgilio, R. 2010. Meta-Modeling of Semantic Web Services. Services Computing (SCC), 2010 IEEE International Conference on.

Downer, C. W. and Ogden, F. L. 2004. "GSSHA: A model for simulating diverse streamflow generating processes." Journal of Hydrologic Engineering **9**(3): 161-174.

Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., and Tompits, H. 2008. "Combining answer set programming with description logics for the Semantic Web." Artificial Intelligence **172**(12-13): 1495-1539.

Euler, L. 1741. "Solutio problematis ad geometriam situs pertinentis (The solution of a problem relating to the geometry of position)." Commentarii academiae scientiarum Petropolitanae: 128-140.

Farrell, J. and Lausen, H. 2007. "Semantic Annotations for WSDL and XML Schema." W3C Recommendation Retrieved October 7, 2010, from http://www.w3.org/TR/sawsdl/.

Favre, J.-M. 2005. Foundations of Model (Driven) (Reverse) Engineering : Models -- Episode I: Stories of The Fidus Papyrus and of The Solarus. Dagstuhl Seminar 04101 on Language Engineering for Model-Driven Software Development, Dagsthul, Germany, Internationales Begegnungs und Forschungszentrum Informatik (IBFI).

Frege, G. 1879. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens (Concept-Script: A Formal Language for Pure Thought Modeled on that of Arithmetic.)* Halle a/S: Verlag von Louis Nebert.

Freitas, F. and Lins, F. 2012. The Limitations of Description Logic for Mathematical Ontologies: An Example on Neural Networks. Proceedings of Joint V Seminar on Ontology Research in Brazil (ONTOBRAS) and VII International Workshop on Metamodels, Ontologies and Semantic Technologies (MOST). A. Malucelli and M. Bax. Recife, Brazil, CEUR-WS. **Vol-938**.

Friedman-Hill, E. 2003. *Jess in Action: Java Rule-Based Systems.* Greenwich, Manning Publications.

Gaševic, D., Djuric, D., and Devedžic, V. 2009. *Model Driven Engineering and Ontology Development.* Berlin, Springer-Verlag.

Gödel, K. 1929. *Über die Vollständigkeit des Logikkalküls (Completeness of the logical calculus).* Ph.D. Dissertation, University of Vienna.

Grau, B. C., Horrocks, I., Kazakov, Y., and Sattler, U. 2008. "Modular reuse of ontologies: Theory and practice." Journal of Artificial Intelligence Research **31**: 273-318.

Grosof, B. N. 2004. SweetRules: Tools for RuleML Inferencing and Translation. 13th Intl. Conf. on the World Wide Web (WWW-2004). New York City.

Grosof, B. N., Horrocks, I., Volz, R., and Decker, S. 2003. Description logic programs: Combining logic programs with description logic. Proceedings of the 12th international conference on World Wide Web. Budapest, Hungary, ACM**:** 48-57.

Hartley, R. T. 1984. CRIB: Computer Fault-finding through Knowledge Engineering. IEEE Computer **17**(3): 76-83.

Hartley, R. T. 1985. Representation of Procedural Knowledge for Expert Systems. Proceedings of the Second IEEE conference on AI applications.

Hebeler, J., Fisher, M., Blace, R., Perez-Lopez, A., and Dean, M. 2009. *Semantic Web Programming.* Indianapolis, Wiley Publishing, Inc.

Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B., and Dean, M. 2004. "SWRL: A Semantic Web Rule Language Combining OWL and RuleML W3C Member Submission 21 May 2004." http://www.w3.org/Submission/SWRL/.

Janssen, S., Andersen, E., Athanasiadis, I. N., and Ittersum, M. K. V. 2008. An European database for integrated assessment and modeling of agricultural systems. iEMSs 2008: International Congress on Environmental Modelling and Software, Barcelona.

Kay, M. 2007 "XSL Transformations (XSLT) Version 2.0." http://www.w3.org/TR/xslt20/.

Klyne, G. and Carroll, J. J. 2004. Resource description framework (RDF): Concepts and abstract syntax., W3C Standard, W3C. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

Kollet, S. J. and Maxwell, R. M. 2006. "Integrated surface–groundwater flow modeling: A free-surface overland flow boundary condition in a parallel groundwater flow model." Advances in Water Resources **29**(7): 945-958.

Lawrence, D. M., Oleson, K. W., Flanner, M. G., Thornton, P. E., Swenson, S. C., Lawrence, P. J., Zeng, X., Yang, Z.-L., Levis, S., Sakaguchi, K., Bonan, G. B., and Slater, A. G. 2011. "Parameterization improvements and functional and structural advances in Version 4 of the Community Land Model." Journal of Advances in Modeling Earth Systems **3**(1): M03001.

Martin, D., Burstein, M., Hobbs, J., Lassila, O., Mcdermott, D., Mcilraith, S., Narayanan, S., Paulucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., and Sycara, K. 2004. "OWL-S: Semantic Markup for Web Services." http://www.w3.org/Submission/OWL-S/.

Masterman, M. 1961. Semantic message detection for machine translation, using an interlingua. International Conference on Machine Translation of Languages and Applied Language Analysis. London, Her Majesty's Stationery Office: 438-475.

Merritt, W. S., Pollino, C., Powell, S., and Rayburg, S. 2009. Integrating hydrology and ecology models into flexible and adaptive decision support tools: the IBIS DSS. 18th World IMACS / MODSIM Congress. Cairns, Australia, MSSANZ.

Motik, B. and Rosati, R. 2007. A Faithful Integration of Description Logics with Logic Programming. International Joint Conference on Artificial Intelligence. Hydrabad, India.

NWIS. 2013. "National Water Information System." http://waterdata.usgs.gov/nwis.

O'Conner, M. J., and Das, A. K. 2009. SQWRL: a Query Language for OWL. OWL: Experiences and Directions (OWLED 2009), Fifth International Workshop. Chantilly, VA.

OGC. 2008. "KML." http://www.opengeospatial.org/standards/kml/.

OMG. 2006. "Meta Object Facility (MOF) Core Specification." http://www.omg.org/mof/

OMG UML. 2010. "OMG Unified Modeling Language(TM) (OMG UML), Infrastructure." http://www.uml.org/

Patel-Schneider, P. F., Hayes, P., and Horrocks, I. 2004. OWL web ontology language semantics and abstract syntax, W3C. http://www.w3.org/TR/owl-features/

Payne, T. and Lassila, O. 2004. "Semantic Web Services." IEEE Intelligent Systems **19**(4): 14-15.

Peirce, C. S. 1885. On the Algebra of Logic. American Journal of Mathematics, **7**(2): 180-196

Piasecki, M., Ames, D., Goodall, J., Hooper, R., Horsburgh, J., Maidment, D., Tarboton, D., and Zaslavsky, I. 2010. Development of an Information System for the Hydrologic Community. 9th International Conference on Hydroinformatics (HIC 2010). Tianjin, China.

Prud'hommeaux, E. and Seaborne, A. 2008. "SPARQL Query Language for RDF." W3C Recommendation. http://www.w3.org/TR/rdf-sparql-query/.

Qu, Y. and Duffy, C. J. 2007. "A semidiscrete finite volume formulation for multiprocess watershed simulation." Water Resources Research. **43**.

Roman, D., Keller, U., Lausen, H., Bruijn, J. D., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., and Fensel, D. 2005. "Web Service Modeling Ontology." Applied Ontology **1**(1): 77-106.

Sattler, U., Baader, F., and Horrocks, I. 2009. Description Logics. *Handbook on Ontologies.* S. Staab and R. Studer, eds. Berlin, Springer-Verlag.

Schindlauer, R. 2008. *Answer Set Programming for the Semantic Web.* Saarbrücken , Germany, VDM Verlag Dr. Müller.

Sirin, E. and Parsia, B. 2004. Pellet: An OWL-DL Reasoner. 2004 International Workshop on Description Logics. V. Haarslev and R. Möller. Whistler, BC, Canada, CEUR-WS.org. **104**.

Sowa, J. F. 1992. Semantic networks. *Encyclopedia of Artificial Intelligence.* S. C. Shapiro, ed. New York, Wiley.

Turing, A. M. 1950. "Computing Machinery and Intelligence." Mind **49**: 433-460.

Villa, F. 2009. Semantically driven meta-modelling: automating model construction in an environmental decision support system for the assessment of ecosystem services

flows. Information Technologies in Environmental Engineering, Springer Berlin Heidelberg**:** 23-36.

W3C. 2010. "SCHEMA." http://www.w3.org/standards/xml/schema.

Wu, J. 2006. Scale and scaling: A cross-disciplinary perspective. *Key Topics in Landscape Ecology.* J. Wu and H. R. Cambridge, eds., Cambridge University Press.

CHAPTER 2

ENABLING APPLIED GEOPHYSICAL CONCEPTUAL AND FUNCTIONAL KNOWLEDGE MODELING: INCORPORATING PROCEDURAL KNOWLEDGE INTO A SEMANTIC REASONING ENGINE[2]

Abstract

One of the grand challenges of the geosciences lies in the automated integration of data sources and computational analyses. The field of knowledge modeling has the potential to enable automated reasoning about data sources – identifying data type, units, and other key information pertinent to computational analyses. While knowledge models are able to represent and reason about concepts, they are limited in their ability to encode and apply procedural knowledge for computational analysis and modeling. This paper presents a new knowledge model representation that integrates conceptual knowledge (e.g. knowledge about data) with procedural knowledge (e.g. procedures, functions or methods). We have termed the new knowledge representation form a “functional ontology” for its capacity to represent and execute procedural tasks as well as reasoning over conceptual knowledge. Underlying the representation of procedural knowledge is a formal definition, which we created, of the logic about the *meaning* of procedural knowledge. This new logic about procedural knowledge has been incorporated into a functional ontology reasoning engine that includes computational procedures as part of its reasoning process. This enables automated connections between data, metadata, and computational analyses. The capability of this system is demonstrated using a simple pedagogical example knowledge model of

---

[2] Prepared for submission to the Journal Computers and Geosciences. The authors are Aaron Byrd and David Tarboton.

polygons and their properties. While seemingly trivial to Geographic Information System (GIS) users, this example demonstrates, at a level where the detail can be exposed, the capability to automatically accomplish GIS style analyses that are beyond the scope of current reasoning engines. While illustrated for a simple GIS example, the functional ontology approach has a generality that is applicable to a wide range of geoscience modeling and data analysis problems. Future work will expand this effort to include a range of computational models, data sources, and tool sets such as watershed delineation tools, and weather, hydrology, and ecological computational models.

2.1 Introduction

Many problems in engineering and physical sciences require an in-depth understanding of a wide variety of both concepts and procedures. For example, in the field of hydrology there are many concepts related to how water moves through the atmosphere, land surface, soil, groundwater, and streams. There are also many procedures that form part of the knowledge base of hydrology, such as how to perform a calculation or execute a model resulting in the quantification of a concept. There are also many concepts and procedures related to creating hydrologic simulations, including information about the data sources, what they mean in relation to watershed model parameters, how to obtain the information, and any processing that must be done to turn the data into simulation input parameters and processing that must be done to generate information in order to deduce conclusions from simulation outputs.

The automated representation, storage, and reasoning of information has long been the scope of the field of artificial intelligence (Turing, 1950). One method, semantic modeling, focuses on developing knowledge models of concepts and the relationships

between them. There are several advantages to using semantic models to represent geophysical knowledge: semantic models provide a common language for scientific interoperability of digital products, semantic models can capture knowledge in a framework that allows for automated, reproducible reasoning, and the reasoning logic capabilities can deduce knowledge not readily apparent, especially with cross-discipline knowledge.

The underlying philosophy of semantic modeling is that "meaning" is entirely a function of the relationships between concepts. Semantic modeling utilizes graph theory (which began with Euler, 1741) to represent the relationships between concepts in order to inform automated reasoning tools how to infer additional relationships between concepts. Undergirding the reasoning tools is a logic called descriptive logic (Brachman, 1979; Ceccato, 1961; Masterman, 1961; Sattler et al., 2009) (see Sowa, 1992) which is derived from first order logic (Frege, 1879; Gödel, 1929; Peirce, 1885).

With the author's background in hydrology, the purpose for examining and utilizing knowledge modeling methods is to develop a methodology that will enable the automation of the functional analyses throughout the process of hydrologic modeling. While many of the concepts and knowledge in hydrology can be represented by an interconnected network of concepts, there is an additional class of knowledge about how to accomplish hydrologic modeling tasks that is in a different spirit, a how-to rather than a what-is. For example, many hydrologic analyses involve analyzing digital land surface elevation models to determine how much land drains to a given point. There have gradually been developed procedures that, with practice, become fairly standard to hydrologists, such as downloading the data, performing the drainage pattern analysis, determining the contributing area and boundaries, etc. These procedures are an important part of the analysis and software such as the

Watershed Modeling System (Aquaveo, 2011), TauDEM (Tarboton et al., 2009), and many others have been created to facilitate this analysis. The procedures we use are directly related to hydrologic concepts and represent vital knowledge. This point, that procedural knowledge is a separate but equally important knowledge base as semantic knowledge, is also argued by (Hartley, 1985).

While there are many types of procedural knowledge, one type that relates to semantic modeling is describing the steps in a functional analysis that relates to how some concepts (e.g. data and metadata) are related. Semantic models (knowledge models built on a reasoning logic) are essentially collections of truthful statements about relationships between concepts. The statements are in the form of "Subject Predicate Object." The relationships between concepts, formed by the Predicate (i.e. verb), are what create the web of concepts and the *meaning* of the concepts.

For example, in Geographical Information System (GIS) analyses frequently one needs to find the area of a polygon. The area can be associated with a polygon identifier in an ontology with a sentence (subject, predicate, object) such as <Polygon_154 hasArea 53.24>, but the method (i.e. code) to determine the area of the polygon could also be a part of the ontology (a "has Area" procedure.) This could be useful if, for example, some external program has not already calculated the area of the polygon. In essence, the code could function as a "live" verb for the predicate. If the ontology is queried for the area of a polygon (<Polygon_154 hasArea ?area>) but the area has not been defined, then the "has Area" procedure could be called (passing the query as a set of parameters) in order to compute the area for the stated polygon. The procedure could then add this new information to the ontology and complete the query.

The purpose of this study is to test the hypothesis that the *meaning* of procedural knowledge can be formalized in order to enable a reasoning engine to execute procedural knowledge as part of a query. The goal is to include procedural knowledge, encoded in a general-purpose programming language, as an integral part of a knowledge base built on semantic models. This will create a general-purpose knowledge modeling language to integrating data, metadata, and functional analyses that we use in the geosciences.

This approach to integrating procedural knowledge into semantic knowledge models is in contrast to, say, utilizing workflow engines for functional analysis. The workflow engines do enable quite complex analyses. The approach to modeling procedural knowledge as a part of a semantic knowledge base, however, has two significant potential advantages. First, the reasoning engine is able to utilize metadata to automatically identify appropriate input data. Secondly, the reasoning logic as well as the procedural knowledge logic enable the reasoning engine to potentially deduce and/or create the required input data.

2.1.1 Semantic Networks and Reasoning

Ontologies are semantic models constructed according to a description logic standard. Description logic standards have a set of keywords that allow the definition of properties of the concepts, such as predicates having a symmetric, transitive, equivalence, class-subclass, etc. relationship. The defined keywords for the description logic standard govern the overall nature of possible relationships between concepts used to express "meaning." These relationships take the form of a simple sentence and can be expressed in a pattern of <Subject Predicate Object>. Much like a sentence, the predicate acts as the verb and defines the relationship between the subject and object. Another way of looking at the

triples is to view them as a "thing-attribute-property" description, where the predicate is the attribute and the property is the object.

The goal of constructing an ontology is to enable a computer to glean and deduce information in answer to a query. Reasoning engine Applications Programming Interfaces (APIs) are built to operate on ontologies and attempt to answer user submitted queries. In essence queries are simple questions asked of the reasoning engine such as <A P ?B> which means "Given a subject **a** and a predicate **p**, what is the set of concepts B (with members $\mathbf{b_i}$) that are true statements <**a p** $\mathbf{b_i}$>." The reasoning engine uses pattern matching and the logic statements to deduce the set of concepts that match the given query parameters. The result is a set of concepts B = {$\mathbf{b_1, b_2, \ldots, b_n}$.}. Another type of query is a truth-test. In this case the query returns whether or not the query triple has been asserted as a truth.

2.1.2 XML and Semantic Modeling

The goal of an ontology is to model concepts. Hopefully this is done in a manner that facilitates modularity and re-use of the ontologies (see Grau et al., 2008). One of the key aspects of making an ontology is to create a set of unique concepts. One of the file formats commonly used is XML and so using Uniform Resource Identifiers (URIs), Uniform Resource Locator (URLs), and XML namespaces are a means of creating and referencing globally unique concepts. The "tag" URL format is used to create globally unique identifiers, such as http://www.example.org/MyOntology.rdf#Concept. Using an XML namespace (ex = http://www.example.org/MyOntology.rdf#) further simplifies the formatting and enhances the human readability of the concept (ex:Concept.) A benefit of using URLs is that there can actually be a web page at the URL that describes the concepts in human terms and as well as host a file to be downloaded and used by a computer. Table 2-1 and Table 2-2 show some

example namespaces, including the namespaces used in the examples and test cases for this research. Resource Description Framework (RDF) (Klyne and Carroll, 2004), Resource Description Framework – Schema (RDF-S or RDFS) (Brickley and Guha, 2004), and Web Ontology Language (OWL) (Patel-Schneider et al., 2004) are description logic languages used in the Semantic Web (Berners–Lee et al., 2001).

2.2 Methods

In order to demonstrate the utility of integrating semantic and procedural knowledge a prototype reasoning engine API was created along with a demonstration knowledge base for computing fundamental properties of polygons, such as the perimeter and area. The approach taken in this API to include procedural knowledge is two-fold. The first method is to define and enable a fall-back mechanism for predicates. In other words, if a query fails to find or deduce a set of concepts matching the query pattern, the predicate function (if that is not the primary query search term) is called to try to create the additional knowledge. The second method to include procedural knowledge is to define and enable execution of a sequence of steps associated with an overarching concept, such as a task the user wants to have performed. In the encoding of the procedural knowledge the API expects actual code. The API compiles and links the code to the appropriate parts of the semantic store.

2.2.1 API Design

The prototype API for the reasoning engine was developed using C# and the code written for the ontology must currently be coded in C# as well. The primary reason for using the C# programming language and the Microsoft .NET framework is that the .NET framework provides a compiler for C# and a method of adding new code to the running code. Since the

code is executed by the operating system rather than interpreted by the semantic model, the reasoning engine must be able to read new code, compile the code, add links to the new code to the semantic model, and then execute the code as required, all without restarting the semantic reasoning engine. If new code is added after other code has already been read, the reasoning engine must be able to remove the current code set and recompile all the code. This is accomplished in the API by wrapping the semantic code into classes for the concepts, compiling it into a library, creating a separate application domain (execution space in memory) for the executable semantic code, loading the library into this domain, and then creating the classes (and references to them) that implement each of the code sections for the concepts.

We have chosen the name “Functional Ontology” for the ontology design, where code can be executed to perform the “how-to” aspects of predicates. It is a new form of attached procedure executable semantic network that links code to semantic reasoning in the two instances described above, namely, query failure and over-arching task concepts. The “functional” aspect of the name does not refer to strictly functional languages, functional programming, or functional (i.e. one-to-one) properties, however, rather to the property of the ontology having executable components that provide specific functionality and represent procedural knowledge.

Table 2-1. Namespaces and Namespace URLs used in this research.

Namespace Symbol	Namespace URL Reference
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
owl	http://www.w3.org/2002/07/owl#
fo	http://chl.erdc.usace.army.mil/FO-lang-20111201#
poly	http://chl.erdc.usace.army.mil/FunctionalOntology/hasAreaExample#

Table 2-2. Description of the namespaces referenced in Table 1-2.

Namespace Symbol	Description
rdf	Resource Description Framework (RDF) is a description logic standard that include some basic logic and the specification of the triple format for storing logic statements. Also specifies an XML file format for ontologies denoted RDF/XML.
rdfs	Builds upon the RDF specification, Resource Description Framework – Schema (RDF-S) includes additional logic for class / subclass relationships
owl	An advanced description logic standard used throughout the semantic web. OWL stands for Web Ontology Language. Includes class / subclass, inverse, equivalence, restrictions, cardinality, and many other types of logic. Often serialized in RDF/XML format.
fo	Functional Ontology description logic standard. Created for the purposes of this research, the functional ontology description logic includes terms specifying how to add algorithms as concepts to an ontology.
poly	Polygon ontology definition. Created for the purposes of this research to contain concepts and algorithms related to polygons and simple GIS functions for polygons.

2.2.2 Functional Ontology Language Definition

A functional ontology will use both existing semantic modeling language keywords as well as extend the keyword list. The current API prototype has a unique description logic that utilizes some keywords from RDF, RDF-S, and OWL along with the RDF/XML file format for serialization. The RDF, RDF-S, and OWL description logic used is meant to demonstrate the reasoning engine capabilities as a proof of concept rather than be a full-fledged description logic implementation.

Since an implementation of a functional ontology in an API will necessitate both the specification of a programming language as well as require compiling the code in the ontology, there will need to be keywords that relate to some of the technical details of the compilation process. Ontologies use namespaces and URIs to create unique identifiers and

represent unique concepts. The namespace chosen for this initial version of the functional ontology keywords is:

fo=http://chl.erdc.usace.army.mil/FO-lang-20111201#

Table 2-3 shows the keywords that are a part of the initial functional ontology namespace. In the functional ontology coding paradigm, the code is organized into the following four primary components:

Primary code refers to a function that can be directly called via a query mechanism. Primary code has a one-to-one relationship with a predicate (e.g. the "has Area" function for the "has Area" predicate).

User code would be code that a user could call from, say, a menu. Instead of simply performing queries against the ontology a user could also call a user function to complete some defined set of actions, such as computing the area for all of the polygons in the ontology.

Secondary code provides a place to write helper methods which can be associated with both primary and user code. It consists of methods (and class members if desired) that are part of the overarching class for primary code or user code, but are outside of the actual function (primary or user) in scope.

Common code is a code base that defines an entire class, minus the class header (for wrapping purposes). Common code allows for the construction of helper classes that can be instantiated by the primary, user, or secondary code. Common code classes are not utilized via an ontological query or direct user calls. In this initial specification class inheritance will not be allowed in order to facilitate wrapping the classes in languages that only allow for single inheritance.

Table 2-3. Key words for associating code with concepts in a functional ontology.

fo:PrimaryCode	The main code for a predicate function. The code should examine the query values, perform the analysis, add the resulting values to the triple store of the ontology, and return the results of the query.
fo:UserCode	The main code for a user-callable function. This code would be run at the user's discretion to run a defined sequence of instructions.
fo:SecondaryCode	This code consists of supporting methods that the primary code or user code could call. The primary code or user code along with the secondary code will be part of a class. Each primary or user code base can have its own secondary code.
fo:CommonClass	This data consists of all the methods and members for a class definition. This code will not be called via a query or user operation but rather it should define class types that can be instantiated by the primary or user code (or their respective secondary code.)
fo:UsingNamespace	In order to include functionality from an external library, often a "using" clause needs to be added to the code. This using namespace value will be added to the individual classes (primary, user, or common) for which they are specified.
fo:UsingFile	This keyword specifies the inclusion of an external library (e.g. system.dll) in the compilation process. The UsingFile keyword specifies the actual library file, while the UsingNamespace keyword specifies a namespace within the library.

In addition to these components, the *UsingNamespace* and *UsingFile* keywords are for use by compilers or interpreters. They direct the compiler to include external files in the compilation process as well as include namespaces from those files in the primary, secondary, and common classes. The "using namespace" commands will be included at the entire ontological class namespace (of which there will only be one for this initial specification) rather than the scope of the individual classes. The using file directives will, of necessity, be of global scope. Duplicate using file and using name directives will be checked and discarded.

2.2.3 Reasoning Algorithm

The API has a prototype reasoning algorithm in order to demonstrate the integration of declarative and procedural knowledge. The reasoning algorithm uses deductive logic to match concept relationships to the query and return the set of concepts that complete the query. The logic is based on RDF and RDF-S but does not include blank nodes or sequences. The reasoning algorithms include the RDF and RDF-S logic for class, subclass, subproperty, domain, range, as well as the logic for the OWL equivalence and inverse of terms.

The reasoning algorithm is built around a recursive search over a triple and node set. Each concept is set up as an instance of a node class with the unique identifier given by the concept string. The node class stores references to the triples of which the node is a part. The recursive algorithm searches for the equivalent nodes and all defined types of each of the concepts that are a part of the query to be matched. For example, if the query is <A B ?C> it looks for all equivalent and types of A and B. From those it then tries to match up all triples that fit the pattern defined by the query. The triples that match the pattern then indicate the set of concepts for the set defined as the missing part of the query; in this case the set of concepts is named “C.” The ontology API has a class defined that holds a dictionary of results sets. The reasoning algorithm creates the results set and returns the count of the concepts in the result set. The reasoning engine follows an open-world paradigm, meaning that null results indicate an unknown rather than a negation. If the query is a truth-test query, meaning that the query is a set of three concepts rather than one or two, the reasoning algorithm returns a 1 if it is known to be true and 0 if it is unknown if it is true.

If the reasoning algorithm does not find a set of concept nodes that fit the query, including for a truth-test query, then it runs the predicate code if it has been defined. The

predicate code is allowed to also run queries and must return the count of concept nodes that fit the query, or the truthfulness of a truth-test query. The predicate code can add triples, initiate other queries (which may also run predicate functions,) and do anything that can be done with the code programming language (C# in the prototype).

2.2.4 Example

The example to demonstrate the utility of the integration of semantic and procedural code will be the evaluation of polygon area. This example is a simple example that is somewhat trivial given that most GIS systems have built-in tools to define the area and perimeter of polygons, but the goal is to demonstrate the potential of integrating procedural knowledge with semantic knowledge and to distill the demonstration to its simplest form. A simple knowledge base about polygons is shown in Figure 2-1.

The “poly” namespace used for this example is:

poly=http://chl.erdc.usace.army.mil/FunctionalOntology/hasAreaExample#

The poly:hasArea and poly:hasPerimeter predicates have predicate functions. The algorithm to find the area of a polygon is not complex. The assumptions here are that the polygon is closed, the line segments connecting the points are linear, and the points are listed in clockwise order with the first point repeated at the end. The area is given by formula 2-1.

$$A = \frac{1}{2}\sum_{i=1}^{n-1}(x_i y_{i+1} - x_{i+1} y_i) \tag{2-1}$$

For the perimeter the Pythagorean distance (in Euclidean two-dimensional space) between each successive point in the list is computed and summed together. For both

functions a listing of the points is required. The point string in the knowledge base is a list of the points in the format "(x_1, y_1) (x_2, y_2) … (x_{n-1}, y_{n-1})." There could certainly be other definitions of points, such as those in the OWL geometry upper level ontology (Dumontier and Gawronski, 2010), but for this example this will be the assumed format.

The code for the knowledge base will actually be comprised of a common class (Figure 2-3) and two predicate procedures (Figure 2-2 and Figure 2-4.) The poly:GetPoints common class (Figure 2-3) will query for the point string and create real-valued arrays for the x and y points, repeating the first point as the last. The poly:hasArea predicate function (Figure 2-2) calls the common class to obtain the points and then computes the area. The poly:hasPerimeter function (Figure 2-4) is similar in format but it computes the perimeter.

The reasoning engine will be run and the ontology for the polygon example case read in. The triples in Figure 2-5 are added to the ontology as the test polygons. Six queries will then be executed, as shown in Table 2-4.

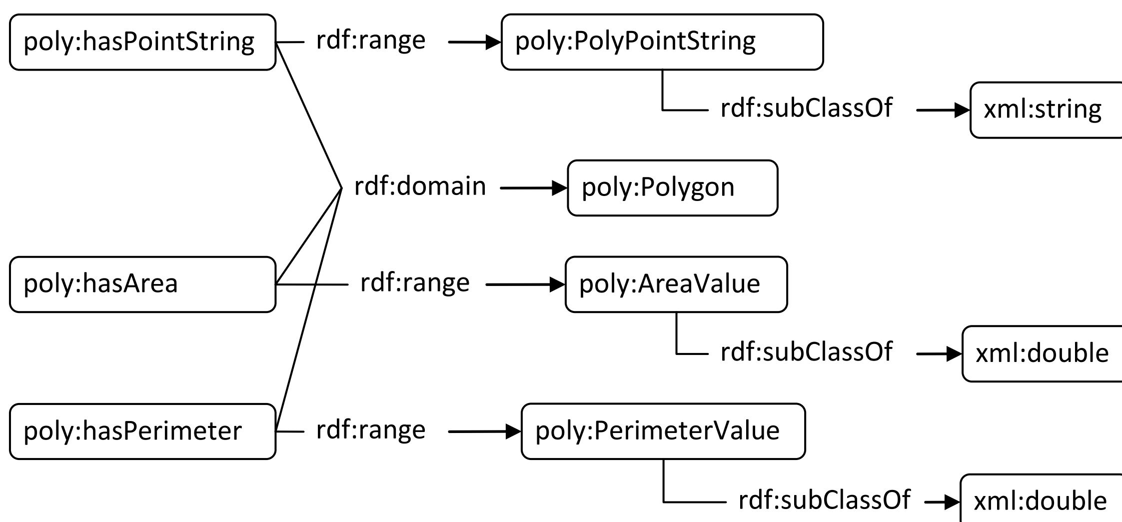


Figure 2-1. Concept map for a simple knowledge base about polygons. It defines polygons as a class and that they may (but not must) have attributes poly:hasPointString, poly:hasArea, and poly:hasPerimeter.

A simple GUI has been created to interface with the functional ontology API.

2.3 Results

With the reasoning engine running the knowledge base defined in the example, and the addition of the triples in Figure 2-5, the reasoning engine is ready for the queries in Table 2-4. Figure 2-6 shows the GUI for the reasoning engine while it is running the functional ontology for the polygon example.

Figure 2-7 shows the reasoning engine GUI after the queries. The results of the queries are the triples shown in Table 2-5 and the named node sets (results sets or answer sets) shown in Table 2-6.

The area and perimeter values for each polygon are not included in the knowledge base and thus each of the queries initially results in an empty set, indicating that it was unable to deduce an answer. The reasoning engine then checked and recognized that "Primary Code" procedural knowledge exists for each of the "poly:hasArea" and "poly:hasPerimeter" predicates. It then executed the method for the primary code for each of the predicates. The primary code for the each of the predicates in turn queries the reasoning engine for the lists of points, calls the common class method to convert the string of points into arrays of double values, computes the desired value, adds the new knowledge to the knowledge base, and re-executes the original query in order to return the new knowledge in the correct format (as part of a set). The final results are new triples added to the knowledge base and the creation of a group of answer sets containing the appropriate values.

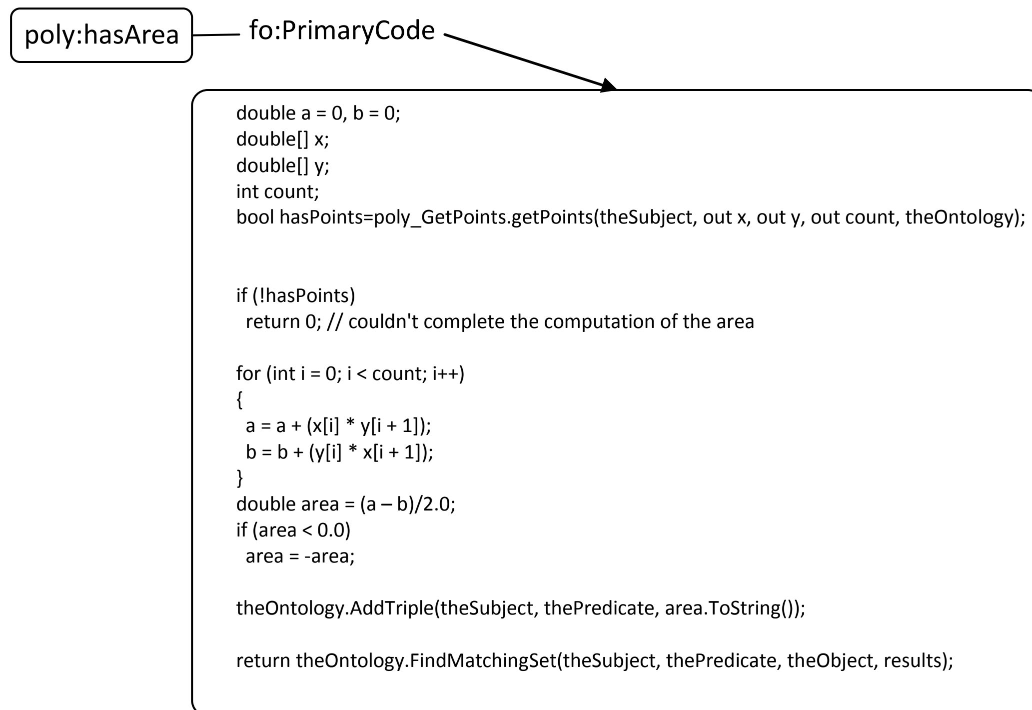


Figure 2-2. The primary code definition for the poly:hasArea predicate. The code here is wrapped in a method for a class with the name of the concept but with the colon replaced by an underscore. The method header is defined by an interface method.

2.4 Discussion

While this example is seemingly simple, the results show that the new reasoning engine was 1) able to execute procedural code describing a non-trivial functional analysis that was 2) encoded in a general-purpose programming language, which 3) enabled the assessment of the existing knowledge in order to deduce knowledge that was not readily apparent. The procedural code was also able to state complex logic that included error checking, data conversion, and mathematical algorithms. The results show that this approach is a viable means of integrating procedural code related to geophysical concepts and semantic querying for geophysical applications involving data conversion, mathematical algorithms, and other computational problems.

The hypothesis that the *meaning* of procedural knowledge can be formalized in order to enable a reasoning engine to execute procedural knowledge as part of a query is thus shown to be true. Procedural knowledge, encoded in a general-purpose programming

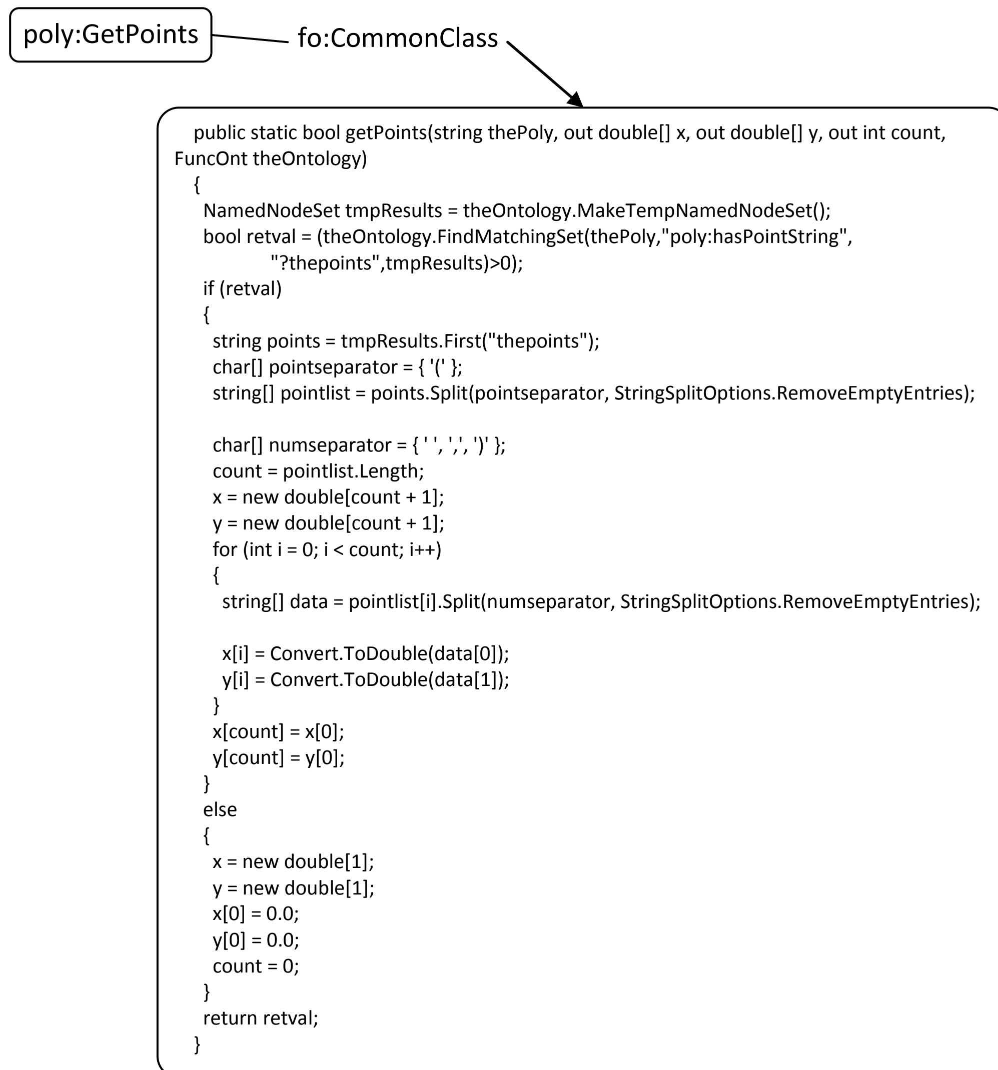


Figure 2-3. The common class definition for the poly:GetPoints utility function to extract points from their storage in a string object.

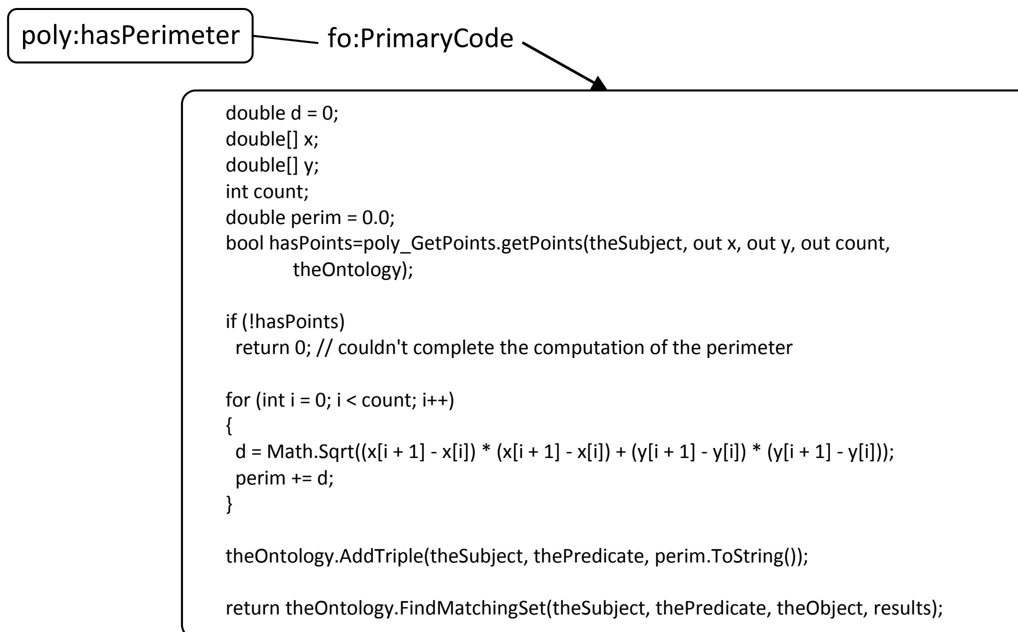


Figure 2-4. The primary code definition for the poly:hasPerimeter predicate, similar to the poly:hasArea predicate primary code.

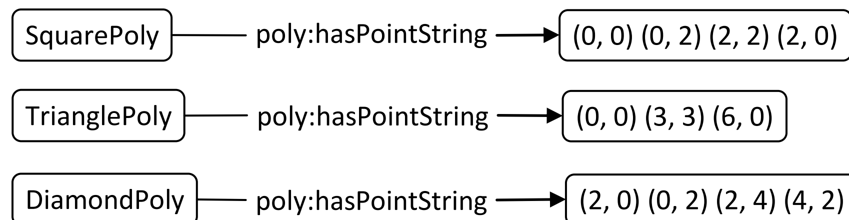


Figure 2-5. Point strings for the test polygons.

Table 2-4. Test queries for the example ontology. These are input into the reasoning engine query mechanism. The objects have a question mark in front of the name indicating that we want concepts that fit there to make true statements. The set of statements is labeled with the name that comes after the question mark, e.g. squareArea.

<SquarePoly	poly:hasArea	?squareArea>
<SquarePoly	poly:hasPerimeter	?squarePerimeter>
<TrianglePoly	poly:hasArea	?triangleArea>
<TrianglePoly	poly:hasPerimeter	?trianglePerimeter>
<DiamondPoly	poly:hasArea	?diamondArea>
<DiamondPoly	poly:hasPerimeter	?diamondPerimeter>

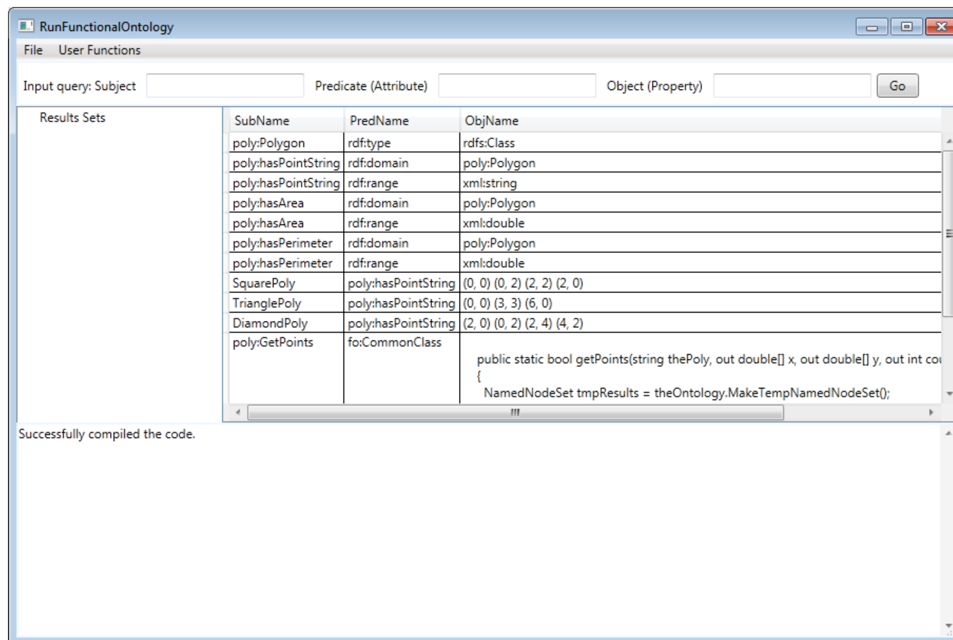


Figure 2-6. The reasoning engine GUI running the functional ontology for the polygon example.

Table 2-5. Triples added to the knowledge base as a result of the code execution via the queries.

Subject	Predicate	Object
SquarePoly	poly:hasArea	4.00
SquarePoly	poly:hasPerimeter	8.00
TrianglePoly	poly:hasArea	9.00
TrianglePoly	poly:hasPerimeter	14.49
DiamondPoly	poly:hasArea	8.00
DiamondPoly	poly:hasPerimeter	11.31

language, is captured as an integral part of a knowledge base built on semantic knowledge models. The prototype reasoning engine is able to utilize both procedural knowledge and semantic reasoning to answer the query posed to it.

As Turing (1950) noted, artificial intelligence algorithms will not create new knowledge but they are able to reach conclusions that human counterparts would not necessarily have thought of. A merit of this approach to including procedural knowledge into a knowledge base is that it enables the reasoning engine to extend its analysis capabilities

and thus reach a broader set of conclusions than previously. This approach does have a drawback in that there is no formal constraint on the logic embedded in the knowledge base. A feature of typical knowledge models is that all conclusions are able to be proven through a rigorous mathematical analysis. The procedural knowledge is a practical “short cut” that enables a wide range of analyses at the expense of also allowing for logical errors and bugs in the code. The one programming the code is responsible for ensuring its logical soundness.

Another merit of this approach to integrating procedural knowledge into a knowledge model is that it constrains subject matter experts to actually state both the concepts and the procedures. This creates a formal procedure for a concept that can be analyzed by other subject matter experts. By creating this formal statement of procedure the art of practice can be studied and improved, turning it into science. For example, in computational modeling a numerical model is chosen to fit a subject purpose. Why one model is chosen over another is generally in the “art” of computational modeling.

Formalizing why we choose one model over another could explain where models overlap and why one model would or wouldn’t be preferred over another. This would push model practitioners to tackle the issues of model applicability and sufficiency, as well as the relationship between cost, accuracy, and expediency.

Table 2-6. Named node sets as a result of the queries. The set members make true statements from the original query.

Set Name	Set Members
SquareArea	{“4.00”}
SquarePerimeter	{“8.00”}
TriangleArea	{“9.00”}
TrianglePerimeter	{“14.49”}
DiamondArea	{“8.00”}
DiamondPerimeter	{“11.31”}

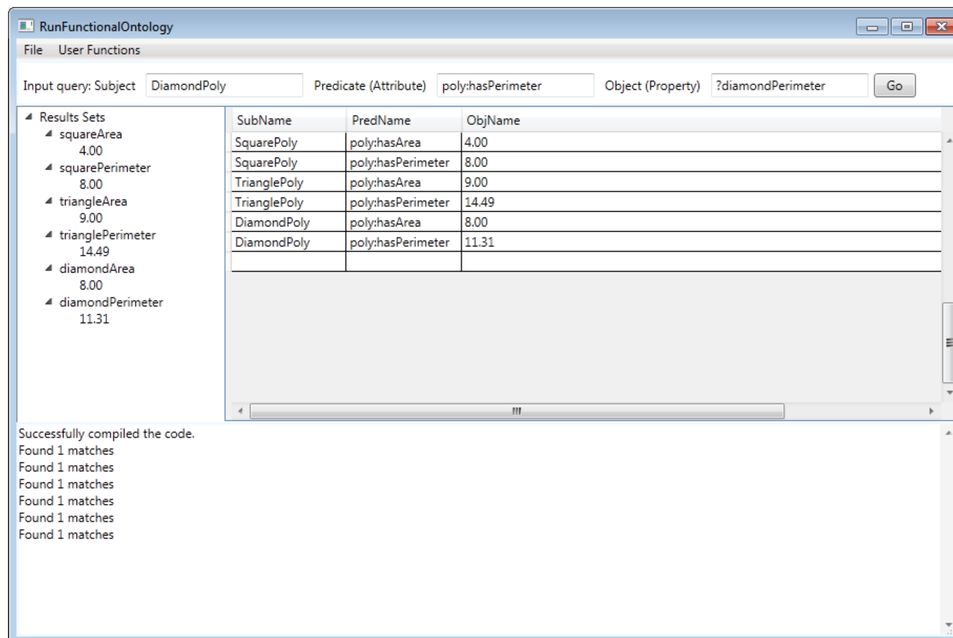


Figure 2-7. The reasoning engine GUI after the results of the query. The resulting sets and the newly added triples are shown.

2.5 Conclusions

Procedural knowledge is very relevant to real-world tasks in the geophysical sciences. The approach to incorporating procedural knowledge in a semantic knowledge base shown in this article, termed “functional ontology,” demonstrates a new, unique, and powerful method of adding procedural execution to a semantic reasoning engine. Through this integration of procedural and semantic knowledge the capabilities of the reasoning engine are greatly enhanced to allow for complex reasoning and analysis of types of problems found in the geophysical sciences.

References

Aquaveo, 2011. Watershed Modeling System (WMS) 8.4. URL: <http://www.aquaveo.com/wms>, [accessed 19 February 2012]

Berners–Lee, T., Hendler, J., Lassila, O., 2001. The Semantic Web. In: Scientific American.

Brachman, R. J., 1979. On the epistemological status of semantic networks. In: Associative Networks: Representation and Use of Knowledge by Computers. N. V. Findler. New York, Academic Press.

Brickley, D., Guha, R. V., 2004. RDF vocabulary description language 1.0: RDF schema, W3C.

Ceccato, S., 1961. Linguistic Analysis and Programming for Mechanical Translation. Gordon and Breach, New York, 242 pp.

Dumontier, M., Gawronski, A., 2010. S. Science. Geometry Ontology. URL: <http://semanticscience.org/ontology/geometry.owl>, [accessed December 7, 2011].

Euler, L., 1741. Solutio problematis ad geometriam situs pertinentis (The solution of a problem relating to the geometry of position). In: Commentarii academiae scientiarum Petropolitanae: 128-140.

Frege, G., 1879. Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens. Halle a. S.

Gödel, K. 1929. Über die Vollständigkeit des Logikkalküls (Completeness of the logical calculus). Ph.D. Dissertation, University of Vienna.

Grau, B. C., Horrocks, I., Kazakov, Y., Sattler, U., 2008. Modular reuse of ontologies: Theory and practice. Journal of Artificial Intelligence Research 31: 273-318.

Hartley, R. T., 1985. Representation of Procedural Knowledge for Expert Systems. Proc. of Second IEEE conference on AI applications.

Klyne, G., Carroll, J. J., 2004. Resource description framework (RDF): Concepts and abstract syntax., W3C. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

Masterman, M., 1961. Semantic message detection for machine translation, using an interlingua. International Conference on Machine Translation of Languages and Applied Language Analysis. London, Her Majesty's Stationery Office**:** 438-475.

Patel-Schneider, P. F., Hayes, P., Horrocks, I., 2004. OWL web ontology language semantics and abstract syntax, W3C. http://www.w3.org/TR/owl-features/

Peirce, C. S. 1885. On the Algebra of Logic. American Journal of Mathematics, 7(2): 180-196

Sattler, U., Baader, F., Horrocks, I., 2009. Description logics. Handbook on Ontologies. S. Staab and R. Studer. Berlin, Springer-Verlag.

Sowa, J. F., 1992. Semantic networks. Encyclopedia of Artificial Intelligence. S. C. Shapiro. New York, Wiley.

Tarboton, D. G., Schreuders, K. A. T., Watson, D. W., Baker, M. E., 2009. Generalized terrain-based flow analysis of digital elevation models. 18th World IMACS Congress and MODSIM09 International Congress on Modelling and Simulation. R. S. Anderssen, R. D. Braddock, L. T. H. Newham. Cairns, Australia, Modelling and Simulation Society of Australia and New Zealand and International Association for Mathematics and Computers in Simulation**:** 2000-2006.

Turing, A. M., 1950. Computing Machinery and Intelligence. Mind 49: 433-460.

CHAPTER 3

ENHANCING HYDROLOGIC DATA CREATION AND PROJECT ANALYSES THROUGH PROCEDURAL AND SEMANTIC MODELING[3]

Abstract

We have many tools and models that we use in hydrologic project analyses. We have different models and tools we use depending on the situation – the project context. For example, planning level studies do not require as much accuracy but do require less time-to-completion than detailed engineering studies. Knowledge models are a class of conceptual models that have the potential of facilitating a wide range of project analyses. By providing a common language for scientific interoperability of digital products, capturing knowledge in a framework that allows for automated, reproducible reasoning, and the reasoning logic to deduce knowledge that it not readily apparent, knowledge modeling holds the promise of enhancing how we use the tools we have. However, to date knowledge models have not been used to discriminate hydrologic work flows based on the project context. Functional ontologies are able to utilize both conceptual knowledge as well as procedural knowledge as part of the knowledge models. We show how combined semantic and procedural modeling can be used to enhance and facilitate existing hydrologic tools. The test case shown in this work covers delineating a watershed with the TauDEM suite of software functions. Semantic models of the TauDEM functions and input and output data are created, along with semantic models of project purposes and how they influence and direct TauDEM analyses. The procedural models include how to construct and execute TauDEM command lines, deduce

[3] Created for publication in the Journal of Environmental Modeling and Software. The authors are Aaron Byrd and David Tarboton.

the chain of dependencies for a given data set, and a user function that, in conjunction with the semantic data, controls and directs the analysis. The result of these knowledge sets is that the reasoning engine is able, because of both the semantic and procedural knowledge, to deduce the chain of functions needed to compute a desired data set. The results show that combined semantic and procedural modeling can enhance and automate the tools we use as hydrologists.

3.1 Introduction

Semantic modeling holds the promise of enhancing how we use the tools we have by providing a common language for scientific interoperability of digital products, capturing knowledge in a framework that allows for automated, reproducible reasoning, and the reasoning logic to deduce knowledge that is not readily apparent. Hydrologic modeling involves a significant amount of analysis and as such semantic modeling would seem to offer advantages, such as reasoning over why and which tools are needed. Semantic models identify concepts and their relationships. These concept relationships form a web or graph of concepts. The underlying philosophy of semantic modeling is that “meaning” of concepts is entirely a function of the relationships between concepts. Semantic modeling utilizes graph theory (which began with Euler, 1741) to represent the relationships between concepts in order to inform automated reasoning tools how to infer additional relationships between concepts. The logic undergirding these reasoning tools is called descriptive logic (Ceccato, 1961; Masterman, 1961; Brachman, 1979; Sowa, 1992; Sattler et al., 2009).

Prior work using semantic models for hydrologic applications includes the Hydrologic Information System (HIS) (Tarboton et al., 2011) created by the Consortium of Universities for the Advancement of Hydrologic Science, Inc. (CUAHSI). CUAHSI HIS facilitates the

discovery and cataloging of hydrologic data. The CUASHSI HIS data discovery and download tool is set up as a web service (Beran et al., 2009). This allows the HIS semantic search and hydrologic data knowledge base to be consumed by programs, such as Hydrodesktop (Ames et al., 2009), and other web services. This situation is typical of the current paradigm for semantic logic programs (knowledge bases built on declarative logic that can include rule sets and conditional logic) and reasoning engines, to use them as a data source consumed by other software.

Reasoning engines have significant potential to facilitate and automate analyses but there remains significant work to be done to make both the knowledge bases and reasoning engines sufficient for the general-purpose analysis and solution of problems. One of the key weaknesses of semantic modeling is that it does not provide a framework for any logic besides the descriptive logic the reasoning engines are built on. The descriptive logic is excellent at providing a formal mechanism for logical deduction but is a poor mechanism for including actual procedural analysis and execution, such as executing a watershed delineation function.

Procedural logic based programs, along with related programming paradigms such as object-oriented programming and functional programming excel at solving specific tasks and encapsulating procedural knowledge. The primary philosophical shortfall of these programming methods, however, is that there is no “meaning” to the procedural logic other than its place in the execution loop. While good programming practices in these domains involve breaking the code into conceptual chunks, these conceptual chunks do not have meaning in a computer-interpretable sense apart from being directly referenced by other portions of the code base. For example, one may write an algorithm that computes the

infiltration of ponded surface water into the soil profile but the computer does not have any relationships about this algorithm that it can use to relate when and why it would need to be called. It only knows to call it when it is invoked by other portions of the code.

The authors have developed a new form of reasoning engine and knowledge base (see Chapter 2) termed “functional ontologies” that aims to help fill that gap in capability by allowing procedural knowledge, represented as source code, to be a part of the semantic knowledge base. The source code, rather than be external to the knowledge base, is included as procedural aspects of the concepts included in the knowledge base. The reasoning engine is able to compile the code and then execute the procedural code as part of a query if need be. The advantage to this approach is that the description of procedural knowledge is in a form that can be readily utilized by the reasoning engine to compute the result without having to have a sophisticated code interpreter. Further, since the form of representation of the procedural knowledge is a general-purpose programming language (C# in the current implementation) the procedural knowledge has the full capabilities of the underlying language. This combination of semantic reasoning and procedural knowledge has the potential of facilitating the automation of many geographic, environmental and hydrologic modeling processes. This article is a demonstration of how it can be used to facilitate the automation of terrain analysis for hydrologic modeling.

For a reasoning engine to be able to successfully replicate and apply the knowledge used for a specific modeling task, here hydrologic terrain analysis, it must be able to both reason about concepts and relationships between them as well as execute established procedures. The hypothesis tested by this work is that the new integrated semantic and procedural knowledge model (functional ontologies) and reasoning engine is capable of

representing the knowledge about how to delineate a watershed, including deducing the workflow and then the execution of the set of actual TAUDem functions, including that for watershed delineation. The overall goal of this work is to explore the practicality of this new form of knowledge representation for use in modeling concepts and procedures involved in geographic, hydrologic and environmental modeling. Success will be quantified by the reasoning engine properly formulating and executing the correct sequence of functions to delineate a watershed based on the query and two separate project contexts.

A reasoning engine that allows for declarative and procedural logic can describe and analyze knowledge about situations. One problem, however, is incorporating existing software and programs. In the field of hydrology there are many existing applications and tools that are used by hydrologists. These applications and tools are built around concepts central to hydrology. Inherent to these tools are relationships between the tools and the datasets they operate on and create – they have, in essence, an implicit ontology that they operate on. The tools do not operate on meaningless data, they operate on specific types of values – stream roughness, soil moisture content, rainfall rates, elevation data, etc. The numbers have a physical meaning. This meaning generally remains in the realm of human interpretation and analysis, for users to understand rather than computers to understand.

The tool sets hydrologists use range in complexity and functional granularity, from a set of command-line executable programs to complex data transformation applications with graphical user interfaces. If the software is able to be controlled programmatically then it has potential for integration with a functional ontology reasoning engine. In order to control the software, however, the knowledge base for the reasoning engine will need to know about

the data sets and functionality available, as well as include functionality for controlling the software.

3.2 Methods

The functional ontology reasoning engine (see Chapter 2) has been designed to integrate semantic and procedural knowledge. In order to demonstrate the utility of modeling semantic and procedural knowledge of existing hydrologic tools, a knowledge model will be created that focuses on using TauDEM (Tarboton et al., 2009) to delineate watersheds. This knowledge model will be a proof-of-concept model designed for watershed delineation rather than the full suite of capabilities of the TauDEM set of functions. The knowledge base will also include the concepts to tailor the TauDEM modeling to the project purpose. There will be two project purposes modeled, one where a single basin delineation is desired for use with engineering-level watershed models and the other where multiple sub-basins are desired for use with planning-level watershed models. A third project purpose, an alternative pit-fill algorithm, is also demonstrated to illustrate how simple it is to add additional analysis end-points.

3.2.1 TauDEM Example: Semantic Knowledge Base

A frequent task undertaken by hydrologists is to create a hydrologic model of a study area that models how and where the water flows under intense rainfall or other flooding conditions. One of the primary tasks when creating a hydrologic model is to delineate the watershed under study. One of the tools used to delineate watersheds is the TauDEM suite of software. We have developed a watershed delineation functional ontology knowledge base that utilizes many of the functional ontology keywords, as well as many RDF (Brickley

and Guha, 2004), RDF-S (Bruijn and Heymans, 2010), and OWL (Patel-Schneider et al., 2004) keywords to encode the knowledge required to run TauDEM. TauDEM consists of a set of command-line executables that begin with a digital elevation model (DEM) file and perform a sequence of analyses to fill digital pits, determine cell-to-cell flow paths and upstream areas, and finally delineate the watershed. Through the processing sequence several data files are created that cover the same raster domain as the original DEM. These files have unique file names that are a derivative of the file name of the original DEM.

This test case is meant to be a somewhat simple proof-of-concept but also to demonstrate and clarify the key capabilities and advances discussed above. While TauDEM has many DEM analysis functions and end-points, such as wetness index analysis, this demonstration will focus on the few functions that are used in watershed delineation. The semantic knowledge base will cover both details about the TauDEM functions and data sets and details relating needed data sets, optional command-line parameter, and the desired data sets for the project purpose. The procedural knowledge base will cover details about how to create TauDEM command lines and execute TauDEM functions.

TauDEM is conceptually a set of command-line executables that have input and output data sets. This relationship between functions and input/output data sets can be described by triples, as shown in Figure 3-1. The use of the owl:InverseOf, rdfs:domain, and rdfs:range keywords are also illustrated in Figure 3-1. The “owl” and “rdfs” part of the keywords refers to their formal definition, which is actually at http://www.w3.org/2000/01/rdf-schema# for RDFS and http://www.w3.org/2002/07/owl# for OWL. The top concept graph defines a type for concepts in the domain and range

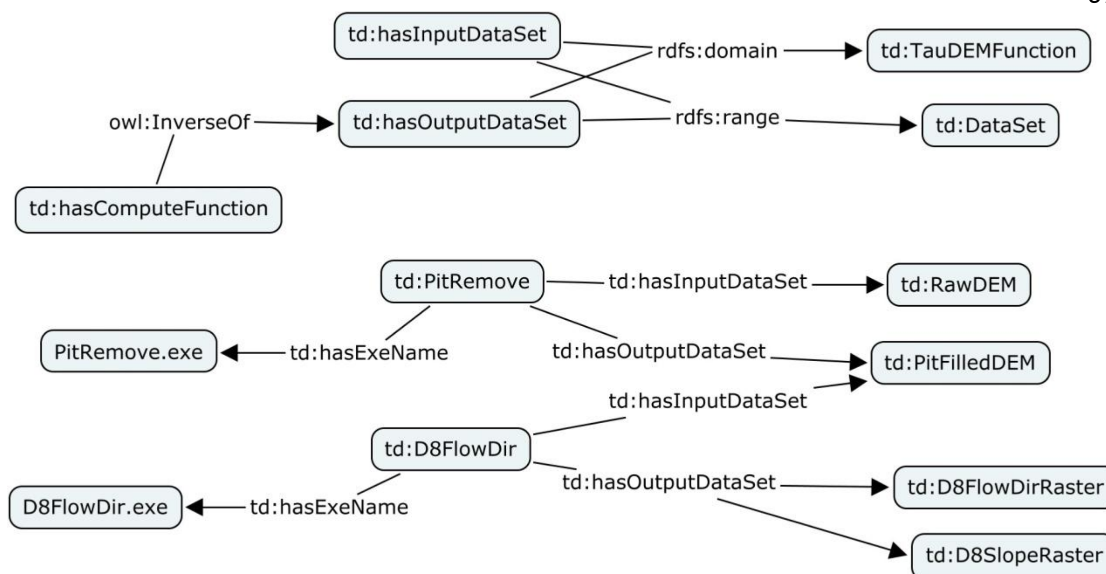


Figure 3-1. The TauDEM triples relating to input and output data sets for the first two functions, the pit remove and 8-way flow direction programs.

positions of the input and output dataset keywords while the second concept graph defines the actual input and output concepts around the first two TauDEM functions, the pit remove and 8-way (D8) flow direction tool. The pit fill function takes as an input a raw DEM and outputs a pit filled DEM, while the D8 flow direction function uses the pit filled DEM as input and outputs D8 flow direction and slope data sets. The graph thus shows an implicit dependence of the D8 flow direction function on the pit remove function.

The owl:InverseOf key word, defined as part of the OWL language, used in Figure 3-1 hasComputeFunction triple <td:hasComputeFunction owl:InverseOf td:hasOutputDataSet>, is used to specify inverses. Mathematically, if it is asserted that <A P B> is true, and P is the inverse of P’, then <B P’ A> is also true. The hasComputeFunction triple, combined with the <td:PitRemove td:hasOutputDataSet td:PitFilledDEM> entails the triple <td:PitFilledDEM td:hasComputeFunction td:PitRemove>; in other words, the Pit Remove function is used to

compute the Pit Filled DEM. The reasoning engine can deduce this triple, and the others like it, when queried.

The TauDEM functions use command-line inputs in order to instruct the executable on what files to read and write. The various data types have specific command-line flags. For example a raw DEM is used in the command line as "-z RawDEMFile.tif." The generated files follow a naming convention, a simple semantics if you will, where a few letters denoting the dataset type are appended to the raw DEM file name. For example a pit filled DEM has the suffix "fel" and, using the previous example, would be denoted in the command line as "-fel RawDEMFilefel.tif." The command line flags and file suffixes for several of the TauDEM data sets are shown in Figure 3-2. The input-only files (i.e. the raw DEM and the outlet files) only have file flags and not file suffixes.

Another set of related ontological keywords, used in Figure 3-2, are "rdfs:subPropertyOf" and "rdfs:SubClassOf." The class/subclass and property/subproperty mechanism serves to create both classes and concepts that are members of another class of concepts, and thus have all the properties of the larger class, while still allowing for unique attributes for the individual subclasses or subproperties themselves. In this example, td:hasInputFileFlag is a subproperty of td:hasFileFlag. Thus, any properties of td:hasFileFlag automatically apply to td:hasInputFileFlag. Similarly for the subclasses td:D8ContribAreaNoOutlet and td:D8ContribAreaAdjOutlet, they inherit the attribute and property "td:hasFileFlag -ad8" from their parent class td:D8ContribArea. While most data sets do not use the input and output file flags, the adjust outlet function takes an outlet shape file and produces an outlet shapefile, thus necessitating the different input and output file flags. When an adjusted outlet is used as an input parameter it is treated as a normal

outlet file, but when it is being output is needs a special suffix and output file flag. The use of the subproperties allows for the td:hasFileFlag to represent both the input and output file flags of most of the data sets, but when specified the input and output flags can be separate.

Figures 3-3 and 3-4 show concept graphs of the TauDEM and terrain group data concepts. These concepts form a logical framework for creating instances of terrain groups that both may and do have TauDEM data sets associated with them. The procedural knowledge, discussed in the following section, details the steps to compute new TauDEM data sets from the given data sets. Figure 3-3 shows concepts and relationships for the terrain data group concept. This knowledge base uses rdfs:domain and rdfs:range, which were discussed above. It also uses rdf:type, which is used to create an instance of a class of concepts. In this case the class of concepts is actually rdfs:Class, which means that the subject concept is a class of concepts. In Figure 3-3 this translates to meaning that td:TerrainDataGroup is a class of concepts and that other concepts can be instances of type td:TerrainDataGroup.

To create the conceptual set of data for a site, an instance of a td:TerrainDataGroup is added to the working ontology, shown in Figure 3-4. The concept :LoganCanyonTerrainGroup and the relationship to the td:TerrainDataGroup concept would be created while interacting with the reasoning engine and not as part of the predefined set of concepts and relationships. The terrain data group :LoganCanyonTerrainGroup, because of the semantic logic for classes, inherits all the properties of the parent class td:TerrainDataGroup, such as the list of potential data sets. The blank namespace in front of LoganCanyonTerrainGroup indicates a local namespace. Figure 3-4 also shows how actual data sets are associated with the instance of the terrain data group. The terrain data group

instance has an associated actual data set, identified by the filename. The type of the data set is also shown.

The knowledge encoded in figures 3-1 to 3-4 comprises a formalization of the knowledge required to use TauDEM. This knowledge is required to be known by a user and is learned by reading the documentation and other writing on watershed delineations. It also formalizes the file naming conventions suggested (but not required) in the TauDEM documentation. This file naming convention is a way of indicating the meaning of the data in the files and as such is an informal ontology. By formalizing the semantics this knowledge representation extends the informal ontology to a formal one that can be utilized for automated processing. Formalizing the semantics also resolves ambiguities and nuances that a human user was just expected to know or learn in order to use TauDEM, such as which data sets serve more than one purpose even though they only have one file name ending denoting their meaning.

3.2.2 TauDEM Example: Procedural Knowledge Base

The TauDEM procedural knowledge base revolves around stepping through the creation sequence of data sets in order to create the desired data set, such as the delineated watershed. Since the API is written in C#, the procedural knowledge is also encoded in C#. For clarity, though, the example code shown in this document simply describes the steps the C# code follows.

The functional ontology specification, http://chl.erdc.usace.army.mil/FO-lang-20111201#, and as discussed in Chapter 2, enables the inclusion of four types of source code: “primary code” used as the “how-to” for predicates (e.g. how to compute an area of a polygon for a “hasArea” predicate,) “user code” which encapsulates procedures for a user-

menu style function (e.g. “compute the area of all polygons”), “secondary code” which are merely helper methods for the primary and user code, and “common classes” where are merely accessory classes for use by the other code. For compatability with C#, there are also keywords to include namespaces and library files in the source code. The reasoning engine takes the data code of the ontology, compiles it, and links it to the appropriate locations in the reasoning engine.

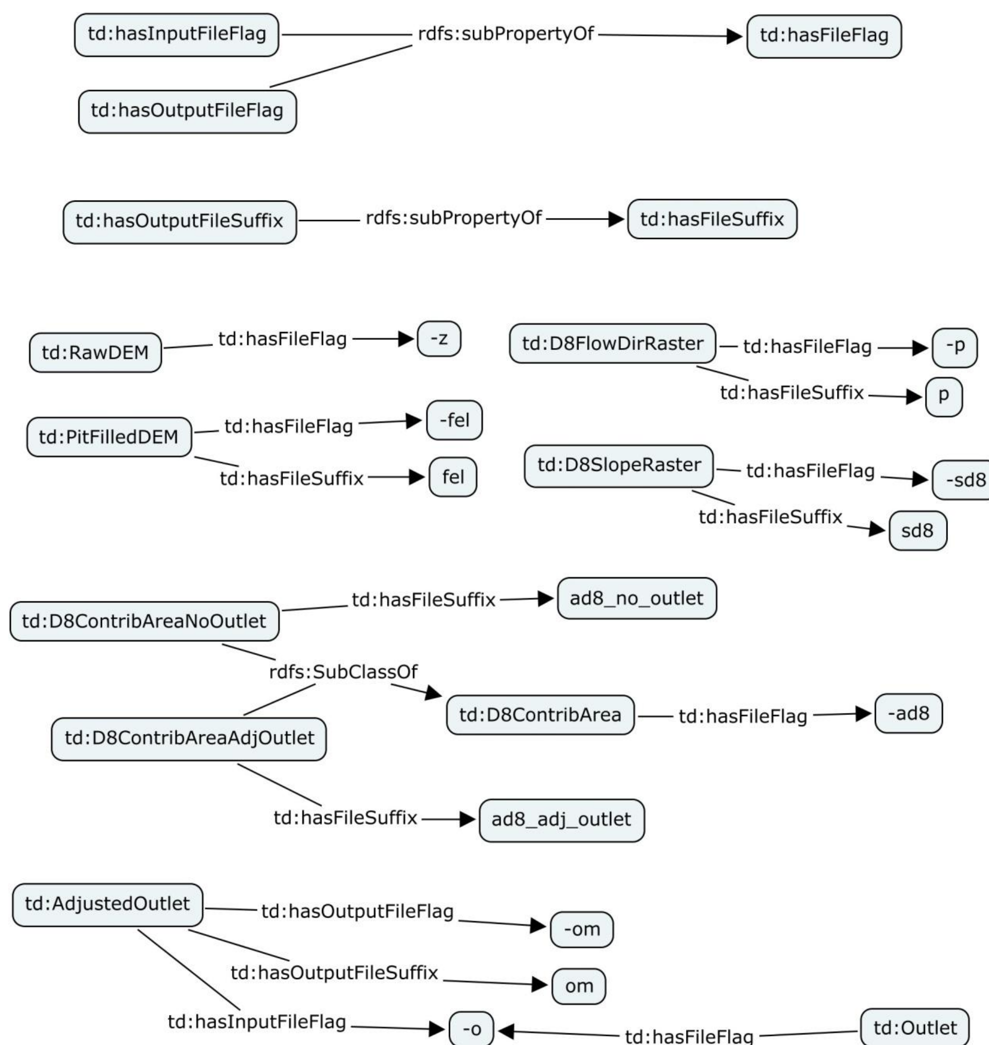


Figure 3-2. Part of the TauDEM knowledge base that utilizes subclasses and sub-properties. These concept graphs show information about the file naming conventions (td:hasFileSuffix) as well as the command-line flags (td:hasFileFlag).

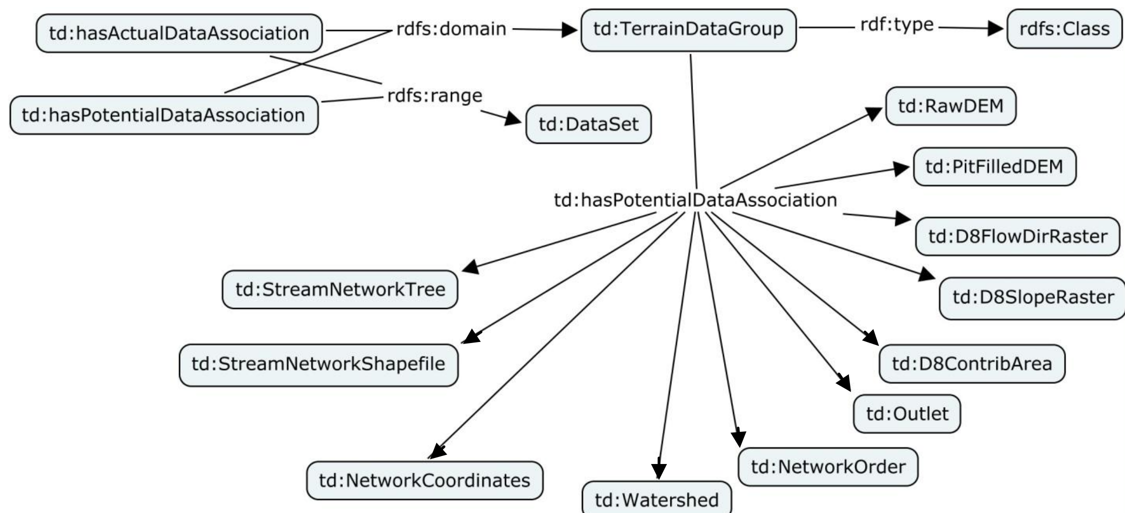


Figure 3-3. The concept map for the terrain data group. The terrain data group is the central concept organizing and framing the set of data sets. The concept td:TerrainDataGroup has potential data associations, while an instance of a terrain data group, e.g. the Logan Canyon data group, would have actual data associations.

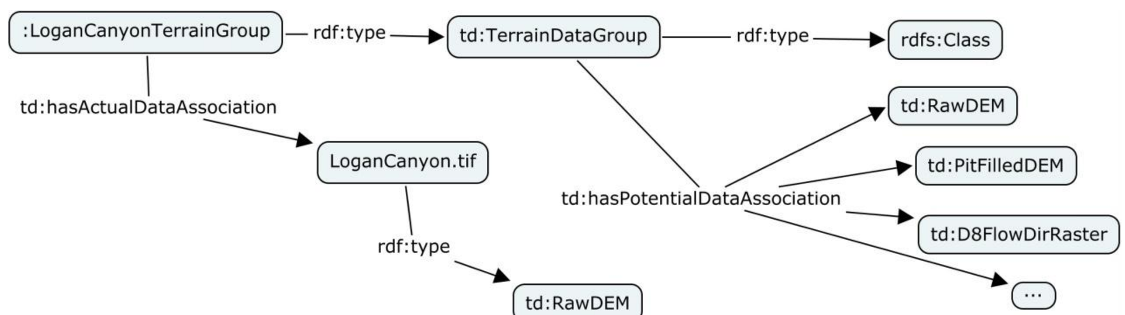


Figure 3-4. Example instance of a terrain data group. The Logan Canyon terrain group is an instance of the class td:TerrainDataGroup. Since td:TerrainDataGroup, as the class definition concept, is associated with several potential data sets LoganCanyonTerrainGroup, via the semantic logic for classes, also has those same potential data sets. Additionally, the LoganCanyonTerrainGroup also has an actual data association LoganCanyon.tif, which is a data set of type td:RawDEM.

The concept graph shown in Figure 3-5, which demonstrates the use of several functional ontology keywords (fo:UsingNamespace, fo:PrimaryCode, and fo:SecondaryCode), contains the knowledge about how to assemble the TauDEM command line. The td:ComputeData and td:hasComputableData concepts have primary code, which is also the code used to complete the semantic query. For example, the query

<:LoganCanyonTerrainGroup td:hasComputableData ?canComputeLogan> will initially result in a function call to the primary code for td:hasComputableData. This code creates a set of concepts that are the currently computable data sets for the given knowledge in the knowledge base. This set is returned as a result set just like any other query would result. But since this knowledge is potentially temporary it is not added to the primary knowledge store. This results in the code being called each time the query is run, thus allowing for the knowledge base and the answer set to evolve over time but still remain a set of true concepts.

The secondary code concept shown in Figure 3-5 is the knowledge that creates the unique file names for the TauDEM data sets. For example, if the root DEM for the terrain data group is LoganCanyon.tif and the computer is trying to create the pit filled DEM, then the GenerateNewFilename code will find the root DEM name (LoganCanyon.tif) and the file name suffix for the new data set ("fel") to compose the new file name. The file name string is split into parts, the suffix appended, and then the parts are reassembled to give LoganCanyonfel.tif. It should be noted that, since this is secondary code, the entire class method is defined in the knowledge base rather than just the code internal to a class method, as is the case for primary code. The secondary code is included as-is as member methods of the class created to house the primary code. Access specifiers (e.g. public/private, static, etc.,) native to the underlying language, control whether or not other external classes are able to access this method. In general, though, secondary code is meant to be used only by the primary code. If it is meant to be more ubiquitously used then it should be created as common code.

Table 3-1 describes the code concepts used to define the TauDEM knowledge base procedural knowledge. It includes several primary code sections, a few common classes, and one user code section. Together this code details how to create terrain data groups, execute a general standard or MPI command-line function, and also how to assemble and execute the TauDEM commands to create a desired data set. Because of the power and flexibility of using code to define the meaning of a predicate, a second-order logic function takes just a few lines of code. Second-order logic refers to logic statements where concepts are linked by an intermediate set of concepts and relationships, rather than being directly related. In this case, the data set dependency list for any given data set is a function of the input data sets for the TauDEM function that outputs the given data set. Thus there is a layer of concepts between the input and output data sets, creating the need for second-order deductive logic.

3.2.3 TauDEM Example: Project Purpose Knowledge Base

In this demonstration example, there are two points where the project is brought in to bear. In the overall analysis of the raw DEM to create the delineated watershed, there are times it is desirable to aim for a watershed with many sub-basins and there are times a single watershed basin is called for. For this demonstration, the distinguishing case will be the user deciding to model a single or multiple basin domain. The single basin example will be presumed to be for a highly-detailed spatially explicit watershed model suitable to analyze watershed management options – an engineering-level. The multiple sub-basin model will be presumed to be for a more tradition hydrologic model that examines rainfall-runoff transformations for existing watershed conditions – a planning level model. Figure 3-6 shows the project purpose relationships, termed the analysis context (i.e. multiple sub-basin model

vs. single-basin model,) to required input data, desired output data, and any special command-line flags for individual TauDEM functions.

To illustrate the ease of adding additional functionality to the ontology, consider Figure 3-7. It illustrates the set of triples required to use the pit remove function to remove digital dams. Digital dams are artifacts of a four-way overland flow algorithm used by some gridded watershed models. When dominant flow directions are lined up at angles askew to the grid arrangement, sometime some cells can block others forming a “digital dam.” Using a four-way pit fill algorithm will ensure that the elevation grid for the watershed model will flow as needed. Because the pit remove function of TauDEM has a four-way as well as the usual eight-way algorithm it can be used for this special circumstance. Figure 3-7 shows the set of nine triples that are required to fully create the option of automatically running the four-way pit fill algorithm as part of the TauDEM functional ontology knowledge base.

The semantic and procedural knowledge base for the TauDEM suite of tools, and the functional ontology API, will be tested on the Logan Canyon DEM and outlet (shown in Figure 3-8) provided as part of the TauDEM example cases. The knowledge base will be run for both the watershed management and rainfall-runoff transformation project purposes.

3.3 Results

Once the TauDEM functional ontology was created, the Functional Ontology Engine (FOE, a graphical user interface designed for use with the functional ontology API) was run with the input RDF XML files. For each of the test cases one command was given: run the user function td:CreateNewTerrainGroup. This user function queries the user for the name of a terrain data group and the purpose of the project. From there the function queries the

ontology for information about the project purpose, including required data and any statements to execute.

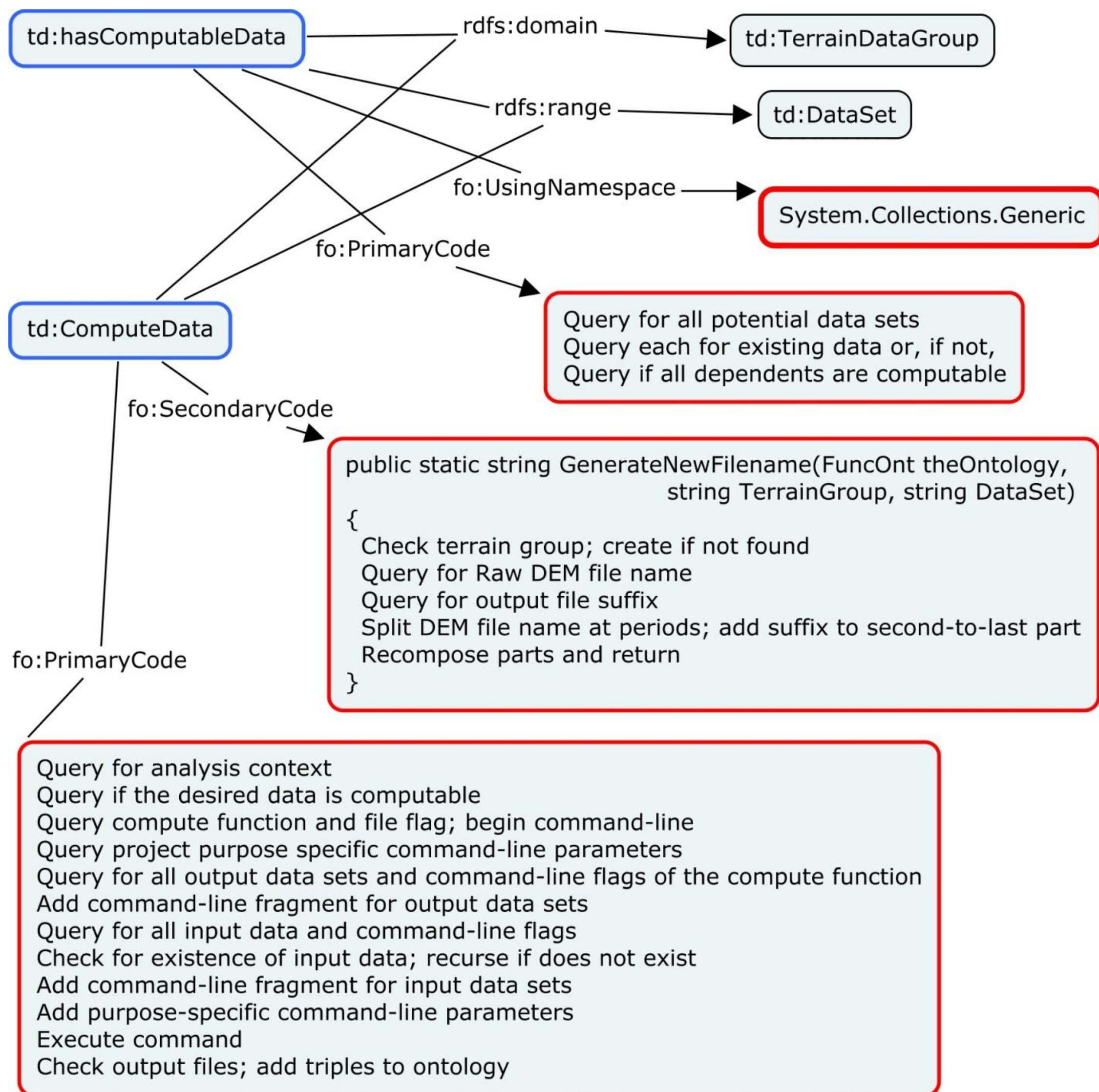


Figure 3-5. Semantic linkages between the previous concept graphs and some of the code definitions. The red boxes show the code that is wrapped in XML CDATA structures when the knowledge base is stored to disk in RDF XML format. The actual code is written in C# whereas this figure describes the process followed by the code.

Table 3-1. Code concepts included in the knowledge base.

Concept	**Description**
td:hasRawDEM (fo:PrimaryCode)	Used in a query to find the root DEM file of a terrain data group. If the terrain data group does not have a raw DEM then it queries td:hasRawDEM the file name.
td:hasRawData (fo:PrimaryCode)	Checks to see if the terrain data group has the specified data; if not the function queries the user for a file name.
td:CreateNewTerrainGroup (fo:UserCode)	Used as a menu option, this function queries the user for the name of the new terrain data group and the purpose of the project. It then queries the ontology for the required data for the project purpose and any statements to execute.
cc:CmdExec (fo:UsingNamespace)	Adds a library namespace used by the code for cc:CmdExec.
cc:CmdExec (fo:CommonClass)	As a common class this code is meant to be used in other functions. The two members are static so that an instance of the class does not have to be created. The two class methods are ExecuteCommand, which executes a standard command in a local shell, and MPIExecuteCommand, which executes a shell command that calls mpiexec to execute the desired command.
cc:DoesFileExist (fo:CommonClass)	Performs a file system query to check the existence of a file. Returns true or false.
cc:FindMPIExecPath (fo:CommonClass)	Checks to see if MPI has been installed by checking the existence of mpiexec in two standard installation locations. If mpiexec is found, it returns the path otherwise it returns an empty string.
td:hasComputeDependence (fo:PrimaryCode)	A second-order logic function, this code is meant to be executed via a query and will return the list of input data sets for the function that outputs the queried data set. Relies on the inverse properties of the 'td:hasComputeFunction' predicate.
td:hasComputableDependents (fo:SecondaryCode)	Defines the function 'DoesTerrainGroupHaveActualData,' which examines the actual data associations of a terrain data group to determine the existence of the specified data set class (e.g. the pit filled DEM.)
td:hasComputableDependents (fo:PrimaryCode)	A recursively defined method (i.e. it queries the ontology with td:hasComputableDependents as a predicate) used to check if all the dependent data sets of the desired data set are computable or exist already.
td:hasComputableData (fo:PrimaryCode)	Searches the list of potential data sets for a given terrain data group to determine which ones are able to

	have all their dependents satisfied given the current actual data associations.
td:hasComputableData (fo:UsingNamespace)	Adds a library namespace for a class used by td:hasComputableData.
td:ComputeData (fo:SecondaryCode)	Defines the method 'GenerateNewFilename,' which queries the for the file suffix of the desired data set and the root DEM of the terrain group to create the filename for the new file according to the customary logic for TauDEM file names. Also includes error checking to make sure the terrain group and root DEM exist.
td:ComputeData (fo:PrimaryCode)	The code that assembles the TauDEM command line (via several queries and calling GenerateNewFilename), executes the command, and adds the filename for the new data to the ontology and the terrain data group. Used in a truth query, it returns success or failure.

Each of the two watershed delineation project purposes need a raw DEM and an outlet (shown in Figure 3-6). Once the files are identified the appropriate triples are added to the ontology to define the new terrain group, the raw DEM file and outlet file, and associate the raw DEM file and outlet file with the terrain group.

Once the required data files are input the execution statement is passed as a query to the reasoning engine. If the query has as one of its parts the concept td:CurrentTerrainDataGroup then the name of the current terrain data group (the one created by the user) is substituted. For both of the project purposes the goal is to delineate a watershed (td:DoWatershedDelineation.) The query (<td:CurrentTerrainDataGroup td:ComputeData td:Watershed>) to delineate the watershed is passed to the reasoning engine.

The compute data function, illustrated in Figure 3-5, queries the ontology for the function that can compute the data set, the input and output data sets of the TauDEM function, and the command line flags for each of them, calls a secondary function to create

new file names for the output data sets, recursively queries the ontology to create dependent data sets that don't yet exist, checks for context assessment functions and adds additional command-line parameters based on the context variables, and then calls some common code to execute the command line in either MPI or standard mode. The compute data command relies on the deductive reasoning engine to return the sets of values it needs to assemble into the command line. For example, the subproperty and subclass reasoning to obtain the correct file suffixes and flags and the inverse-of reasoning to obtain the TauDEM function that can compute the desired data sets. The compute data function has 18 different calls to the reasoning engine, several of which often call other functions in turn as well as recursively calling the compute data function to compute dependent data sets that do not exist yet.

For this exercise both watershed delineation analysis contexts were used as two different trials. The single-basin result is shown in Figure 3-9 (a) and the multiple-basin result in shown in Figure 3-9 (b). The triples in the ontology created through this process, for the single-basin analysis context, are shown in Table 3-2. The multiple-sub-basin analysis context is similar. Figure 3-10 shows an excerpt from the output of the reasoning engine.

3.4 Discussion

The key goal of a functional ontology is to include programming language code as part of a knowledge base, and to include it in a fashion a reasoning engine is able to utilize and to know when to utilize. The results of the demonstration show how integrated semantic reasoning and procedural execution can enhance tradition hydrologic analyses, as well as how the project situation can influence can be presented to influence the analyses.

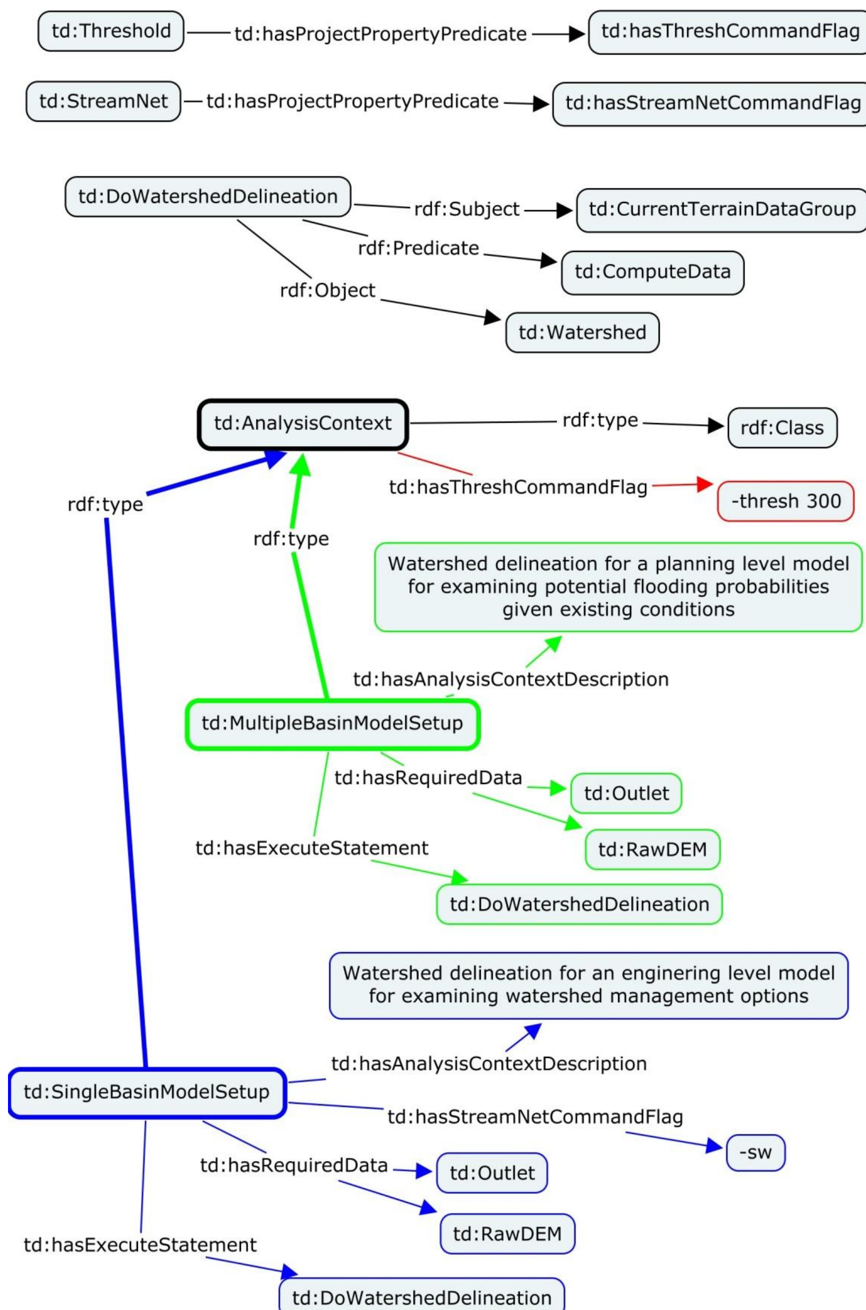


Figure 3-6. The analysis context (project purpose). The analysis context sets the overall big picture of the desired data. The required data must be input by the user; the execute statement is the goal of the analysis. TauDEM functions can define specialized command flags for individual analysis contexts (td:MultipleBasinModelSetup or td:SingleBasinModelSetup,) such as td:hasStreamNetCommandFlag, or for the group of analysis contexts via the semantic logic (such as the td:hasThreshCommandFlag shown in red). The colors highlight the two different knowledge base sets for the two project purposes.

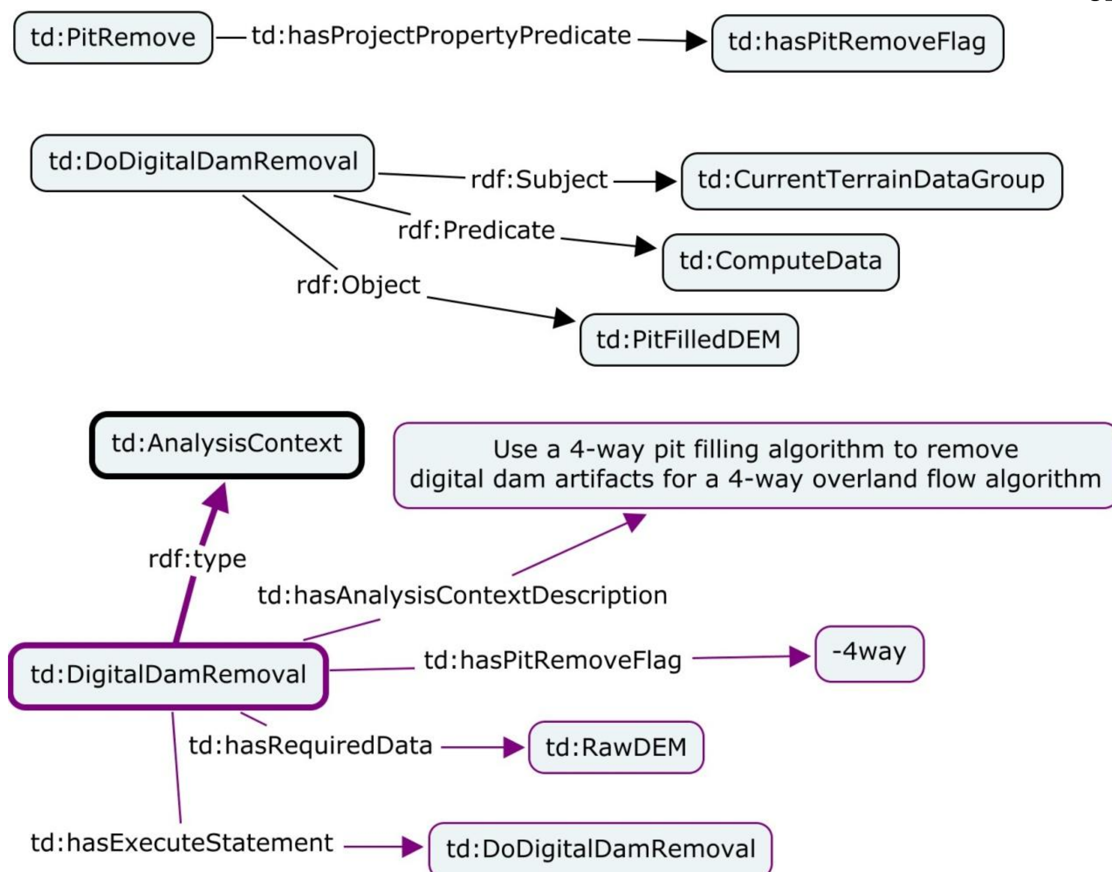


Figure 3-7. Example of an additional analysis context. This analysis context uses a 4-way pit filling algorithm to solve digital dam problems.

The procedural code (see Figure 3-5 and Table 3-1) executed queries against the semantic triples. The semantic queries used declarative logic in finding matches to the queries and, when needed, executed the predicate procedures. The predicate procedures themselves executed queries in order to recurse through the list of dependent to a data set. The general-purpose ability of the procedural knowledge to enhance the reasoning logic sets this knowledge model apart from other knowledge modeling forms. The use of concepts, concept relationships, and deductive reasoning logic sets this method of knowledge modeling apart from scripting languages and many other programming forms.

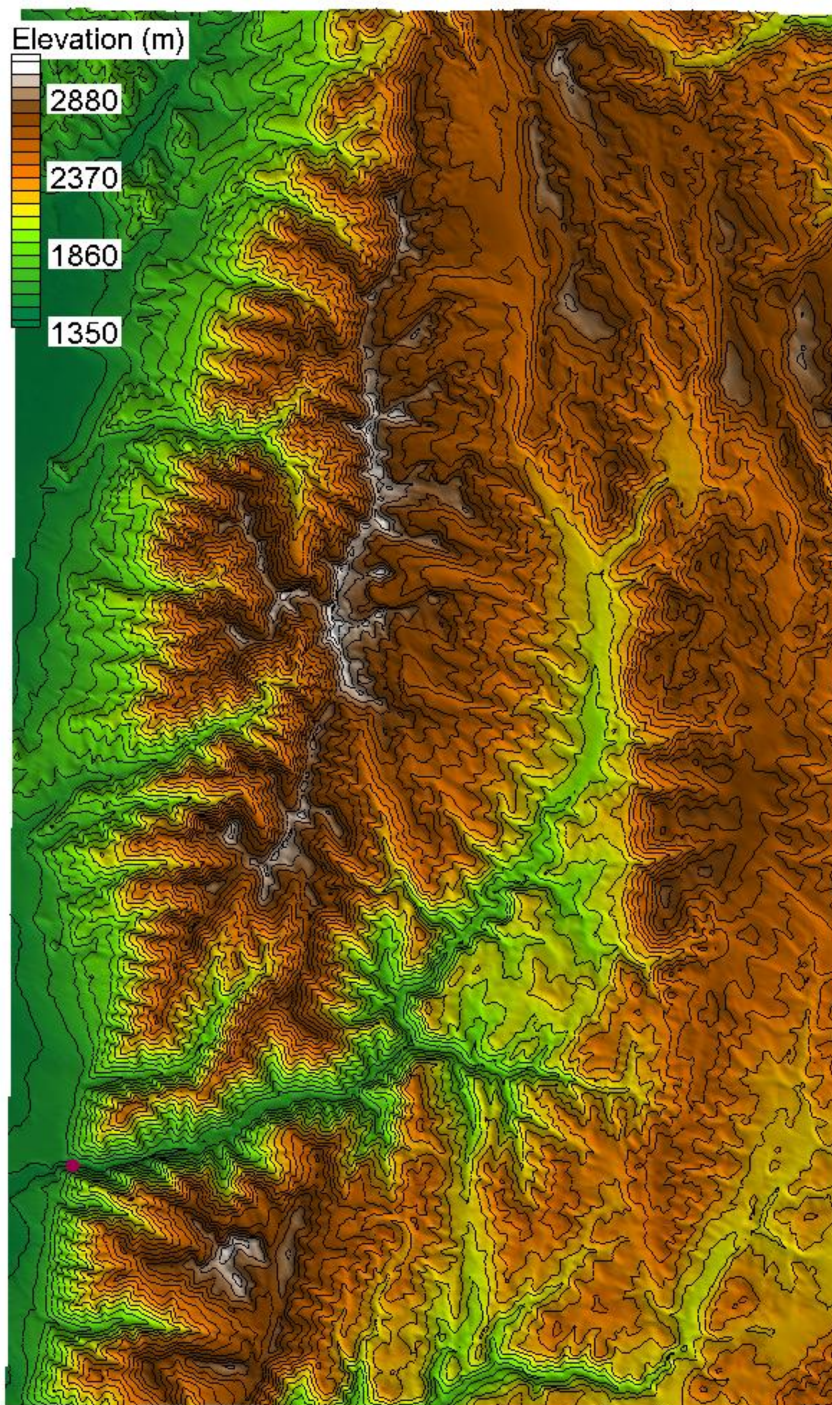
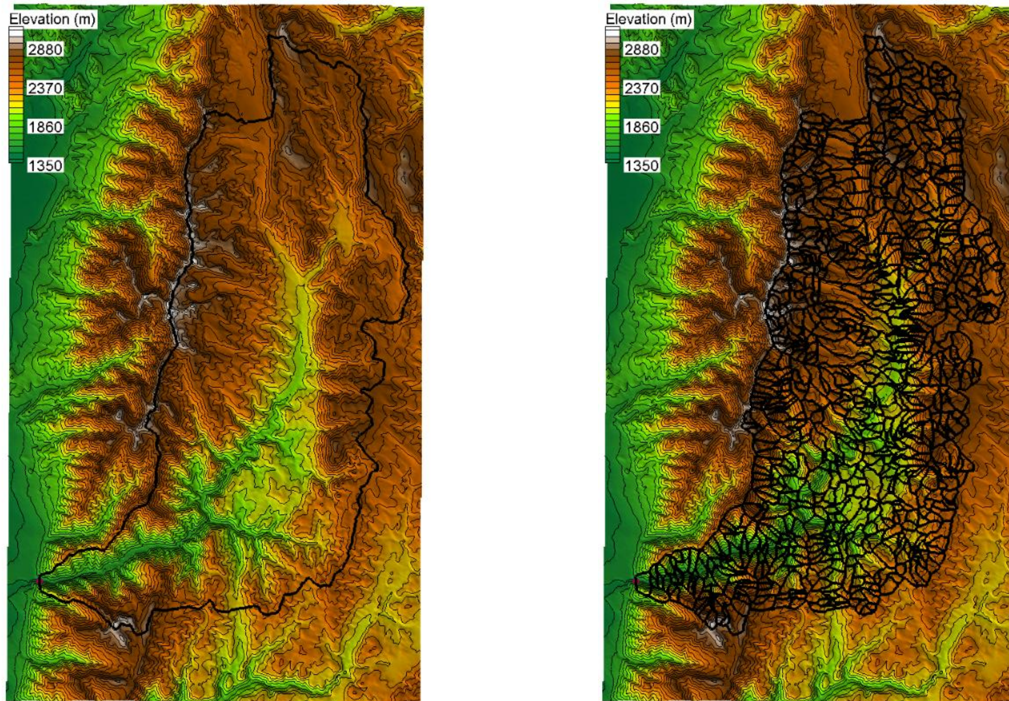


Figure 3-8. Contours of the raw DEM file used inthe demonstration example, LoganCanyon.tif. The pink dot near the lower left corner is the outlet location (LoganOutlet.shp).



(a) Single basin watershed

(b) Multiple basin watershed

Figure 3-9. Watershed basins produced by the TauDEM suite of tools, the TauDEM knowledge base, and the functional ontology API.

Having the code as an active part of the knowledge base creates the opportunity for discussion of the code among a community of software engineers who specialize in the focus of the ontology. The analysis approach is made bare for all to view and discuss. This paradigm allows for an expert systems approach whereby both the knowledge about system concepts as well as analysis methods are encoded into a “living” document that the computer can operate on to conduct independent analyses. The functional aspects of the analysis are not hidden behind a user interface or buried in a set of code files but rather are an integral part of the input to the system. Each code snippet is associated with a concept and can employ all the expressive power of the coding language to fully create the meaning of the action behind the concept.

Table 3-2. Semantic triples created by the execution of the td: CreateNewTerrainGroup user function, as discussed above. The analysis context used was the single basin model.

Subject	Predicate	Object
:LoganCanyonTerrainGroup	rdf:type	td:TerrainDataGroup
LoganCanyon.tif	rdf:type	td:RawDEM
:LoganCanyonTerrainGroup	td:hasActualDataAssociation	LoganCanyon.tif
LoganOutlet.shp	rdf:type	td:Outlet
:LoganCanyonTerrainGroup	td:hasActualDataAssociation	LoganOutlet.shp
:LoganCanyonTerrainGroup	td:hasActualDataAssociation	LoganCanyonfel.tif
LoganCanyonfel.tif	rdf:type	td:PitFilledDEM
LoganCanyonfel.tif	td:hasPitFillAlgorithm	td:D4Planchon
:LoganCanyonTerrainGroup	td:hasActualDataAssociation	LoganCanyonp.tif
LoganCanyonp.tif	rdf:type	td:D8FlowDirRaster
:LoganCanyonTerrainGroup	td:hasActualDataAssociation	LoganCanyonsd8.tif
LoganCanyonsd8.tif	rdf:type	td:D8SlopeRaster
:LoganCanyonTerrainGroup	td:hasActualDataAssociation	LoganCanyonad8_no_outlet.tif
LoganCanyonad8_no_outlet.tif	rdf:type	td:D8ContribAreaNoOutlet
:LoganCanyonTerrainGroup	td:hasActualDataAssociation	LoganCanyonsrc_no_outlet.tif
LoganCanyonsrc_no_outlet.tif	rdf:type	td:StreamRasterNoOutlet
:LoganCanyonTerrainGroup	td:hasActualDataAssociation	LoganCanyonom.shp
LoganCanyonom.shp	rdf:type	td:AdjustedOutlet
:LoganCanyonTerrainGroup	td:hasActualDataAssociation	LoganCanyonad8_adj_outlet.tif
LoganCanyonad8_adj_outlet.tif	rdf:type	td:D8ContribAreaAdjOutlet
:LoganCanyonTerrainGroup	td:hasActualDataAssociation	LoganCanyonsrc_adj_outlet.tif
LoganCanyonsrc_adj_outlet.tif	rdf:type	td:StreamRasterAdjOutlet
:LoganCanyonTerrainGroup	td:hasActualDataAssociation	LoganCanyonord.tif
LoganCanyonord.tif	rdf:type	td:NetworkOrder
:LoganCanyonTerrainGroup	td:hasActualDataAssociation	LoganCanyontree.dat
LoganCanyontree.dat	rdf:type	td:StreamNetworkTree
:LoganCanyonTerrainGroup	td:hasActualDataAssociation	LoganCanyoncoord.dat
LoganCanyoncoord.dat	rdf:type	td:NetworkCoordinates
:LoganCanyonTerrainGroup	td:hasActualDataAssociation	LoganCanyonnet.shp
LoganCanyonnet.shp	rdf:type	td:StreamNetworkShapefile
:LoganCanyonTerrainGroup	td:hasActualDataAssociation	LoganCanyonw.tif
LoganCanyonw.tif	rdf:type	td:Watershed
LoganCanyonw.tif	td:AnalysisContext	td:4PointFlowModelSetup
LoganCanyonw.tif	td:hasBasinCount	td:SingleBasin

The tight integration of the general purpose procedural code with the semantic network and semantic reasoning engine constitute a new paradigm of computer programming. It is not strictly a declarative logic programming style nor is it a strict procedural logic programming style but rather a blending of the two. Functional ontologies allow programmers to reference globally accepted concepts and tie code to these concepts.

```
Compiling code to
          C:\Users\Administrator\AppData\Local\Temp\y2i0nqxw.1ta\fos_364e8f13-7f65-
          4257-8e1c-52d1db6cee79.dll
Successfully compiled the code.
Checking CA td:useMPI
Set default value (td:SingleProcessor) for td:useMPI
Executing statement <LoganCanyonTG td:ComputeData td:Watershed>
Found 15 computable data sets, 2 actual data sets, and 0 uncomputable data sets.
Determining command-line format of the 5 output data sets for td:StreamNet
The reasoning engine deduced that there are 5 output data sets, td:NetworkOrder
          (c:\work\TestProjects\LoganCanyon\LoganCanyonord.tif) td:StreamNetworkTree
          (c:\work\TestProjects\LoganCanyon\LoganCanyontree.dat) td:NetworkCoordinates
          (c:\work\TestProjects\LoganCanyon\LoganCanyoncoord.dat)
          td:StreamNetworkShapefile
          (c:\work\TestProjects\LoganCanyon\LoganCanyonnet.shp) td:Watershed
          (c:\work\TestProjects\LoganCanyon\LoganCanyonw.tif)
Determining input data sets for td:Watershed
Dependent data set td:PitFilledDEM not computed, recursing via the reasoning engine...
Found 15 computable data sets, 2 actual data sets, and 0 uncomputable data sets.
Determining command-line format of the 1 output data sets for td:PitRemove
The reasoning engine deduced that there is 1 output data set, td:PitFilledDEM
          (c:\work\TestProjects\LoganCanyon\LoganCanyonfel.tif)
Determining input data sets for td:PitFilledDEM
Deduced that the correct command line is: PitRemove.exe  -fel
          c:\work\TestProjects\LoganCanyon\LoganCanyonfel.tif  -z
          c:\work\TestProjects\LoganCanyon\LoganCanyon.tif
[It would be nice if the log reported successfully executing a function]
Successfully created the new td:PitFilledDEM data set
          c:\work\TestProjects\LoganCanyon\LoganCanyonfel.tif.
Successfully computed the data set td:PitFilledDEM
Dependent data set td:D8FlowDirRaster not computed, recursing via the reasoning
          engine...
Found 14 computable data sets, 3 actual data sets, and 0 uncomputable data sets.
Determining command-line format of the 2 output data sets for td:D8FlowDir
The reasoning engine deduced that there are 2 output data sets, td:D8FlowDirRaster
          (c:\work\TestProjects\LoganCanyon\LoganCanyonp.tif) td:D8SlopeRaster
          (c:\work\TestProjects\LoganCanyon\LoganCanyonsd8.tif)
Determining input data sets for td:D8FlowDirRaster
…
```

Figure 3-10. Excerpt from the log produced by running the td:CreateNewTerrrainDataGroup user method and choosing the single-basin project purpose.

One of the potentials benefits of this approach is that general purpose algorithms can be researched and constructed to mine data, deduce further information from the data, and also potentially find inconsistencies with the algorithms and reality, thus aiding the identification of research opportunities.

Further, the functional ontology approach also is a step towards realizing the fourth paradigm of discovery, data-intensive discovery (Hey et al., 2009) wherein computers do the bulk of the analysis and discovery on huge data sets, guided by the concepts and understanding given by the experts. Functional ontologies should provide a mechanism to facilitate the automated processing and assessment of large quantities of data in a manner that could yield results in a fraction of the time in which it could be done via human interaction. This data-to-theory paradigm has a competing paradigm, a tools-to-theory paradigm (Gigerenzer, 2000) that posits that the way to increased understanding relies on increasingly advanced tools to research and measure information about the world. There is likely a feed-back mechanism in the creative development of new tools for measuring data corroborating theories. For hydrology, the ultimate goal is then to have a tool that can, in an automated fashion, research information about a watershed, evaluate the appropriateness of the modeling tools, and then create an analysis model that accounts for both user purposes and the on-site processes.

An alternative approach for scripting actions are workflows such as those in ArcGIS ModelBuilder (Armstrong, 2009). There are two disadvantages of using workflows when compared to using functional ontologies. First, the functional ontology approach can utilize information about the data and models to discover which data and models should be applied. Secondly, by having the reasoning engine querying for the data rather than having

the user supply the data the reasoning engine could use other procedural knowledge to create the data set if needed.

3.5 Conclusions

The results demonstrate that artificial intelligence reasoning engines can be successfully utilized for hydrologic purposes. The demonstration showed how an integrated knowledge base of TauDEM concepts (declarative knowledge), procedures relating to the execution of TauDEM functions (procedural knowledge), and relationships between the analysis context and the optional TauDEM execution parameters (contextual declarative and procedural knowledge) can be use to analyze data, create 15 data sets, and in the end delineate a watershed, all in an automated manner.

The integration of these forms of knowledge resulted in a knowledge system that is more capable than either one alone. Through the modeling of concepts and procedures related to those concepts, functional ontologies could play a significant role in the advancement of the state of science of automated information processing and modeling.

References

Ames, D.P., Horsburgh, J., Goodall, J., Whiteaker, T., Tarboton, D., Maidment, D., 2009. Introducing the Open Source CUAHSI Hydrologic Information System Desktop Application (HIS Desktop). In: Anderssen, R.S., Braddock, R.D., Newham, L.T.H. (Eds.), 18th World IMACS Congress and MODSIM09 International Congress on Modelling and Simulation. Modelling and Simulation Society of Australia and New Zealand and International Association for Mathematics and Computers in Simulation: Cairns, AU, pp. 4353-4359.

Armstrong, K., 2009. ModelBuilder-An Introduction, 2009 ESRI Southeast Regional User Group Conference. ESRI: Jacksonville, FL.

Beran, B., Goodall, J., Valentine, D., Zaslavsky, I., Piasecki, M., 2009. Standardizing Access to Hydrologic Data Repositories through Web Services. Proceedings of the International

Conference on Advanced Geographic Information Systems & Web Services (GEOWS 2009). IEEE Computer Society: Los Alamitos, CA, pp. 64-67.

Brachman, R.J., 1979. On the epistemological status of semantic networks. In: Findler, N.V. (Ed.), Associative Networks: Representation and Use of Knowledge by Computers. Academic Press: New York.

Brickley, D., Guha, R.V., 2004. RDF vocabulary description language 1.0: RDF schema. W3C. http://www.w3.org/RDF/

Bruijn, J.d., Heymans, S., 2010. Logical Foundations of RDF(S) with Datatypes. Journal of Artificial Intelligence Research 38, 535-568.

Ceccato, S., 1961. Linguistic Analysis and Programming for Mechanical Translation. Gordon and Breach, New York.

Euler, L., 1741. Solutio problematis ad geometriam situs pertinentis (The solution of a problem relating to the geometry of position). Commentarii academiae scientiarum Petropolitanae 128-140.

Gigerenzer, G., 2000. Adaptive Thinking: Rationality in the Real World. Oxford University Press, New York.

Hey, T., Tansley, S., Tolle, K., 2009. The Fourth Paradigm: Data-Intensive Scientific Discovery. Microsoft Research, Redmond.

Masterman, M., 1961. Semantic message detection for machine translation, using an interlingua, International Conference on Machine Translation of Languages and Applied Language Analysis. Her Majesty's Stationery Office: London, pp. 438-475.

Patel-Schneider, P.F., Hayes, P., Horrocks, I., 2004. OWL web ontology language semantics and abstract syntax. W3C. http://www.w3.org/TR/owl-features/

Sattler, U., Baader, F., Horrocks, I., 2009. Description logics. In: Staab, S., Studer, R. (Eds.), Handbook on Ontologies. Springer-Verlag: Berlin, pp. 21-44.

Sowa, J.F., 1992. Semantic networks. In: Shapiro, S.C. (Ed.), Encyclopedia of Artificial Intelligence, 2nd ed. Wiley: New York. http://www.jfsowa.com/pubs/semnet.htm

Tarboton, D.G., Maidment, D., Zaslavsky, I., Ames, D., Goodall, J., Hooper, R.P., Horsburgh, J., Valentine, D., Whiteaker, T., Schreuders, K., 2011. Data Interoperability in the Hydrologic Sciences, The CUAHSI Hydrologic Information System, In: Jones, M.B., Gries, C. (Eds.), Proceedings of the Environmental Information Management Conference 2011. University of California: Santa Barbara, CA, pp. 132-137.

Tarboton, D.G., Schreuders, K.A.T., Watson, D.W., Baker, M.E., 2009. Generalized terrain-based flow analysis of digital elevation models, In: Anderssen, R.S., Braddock, R.D.,

Newham, L.T.H. (Eds.), 18th World IMACS Congress and MODSIM09 International Congress on Modelling and Simulation. Modelling and Simulation Society of Australia and New Zealand and International Association for Mathematics and Computers in Simulation: Cairns, Australia, pp. 2000-2006.

CHAPTER 4

FACILITATING COMPUTATIONAL MODEL INTEGRATION WITH SEMANTIC AND PROCEDURAL KNOWLEDGE MODELS[4]

Abstract

Given that water is a driving force in many environmental and ecological situations, using hydrologic models together with environmental or ecological models for decision purposes is happening more frequently. There are many challenges in model integration but two key challenges addressed by this work are identifying models that are able to provide the data of appropriate type and sufficient scale and using models with unique input and output data formats and execution mechanisms as a part of an integrated system. Semantic modeling of procedural computations provides a new approach to facilitate model integration. By utilizing metamodels, or models of models, a semantic reasoning engine can reason about integrating disparate and unique models. The other key to model integration is to view the process not as an attempt to integrate models but rather a process of deducing one form of data from another. In order to use the reasoning engine to deduce relationships between data sets a distinction must be made between fully mutable, partially mutable, and immutable properties of data. Similar to the requirement for semantic models to use unique concept identifiers, the immutable and partially mutable properties of data sets act as a unique fingerprint that procedural and semantic knowledge can use to identify them. Using this data identification and computational metamodel approach together results in a

[4] Created for publication in the Journal of Environmental Modeling and Software. The authors are Aaron Byrd and David Tarboton.

functional ontology model integration system that can use disparate models in a coherent fashion, automatically selecting the models to use.

In order to demonstrate how semantic and procedural modeling can facilitate model integration, a demonstration project predicting climate change impacts for a study area in Georgia has been created. Two climate change scenarios are downscaled and turned into sets of weather data. A hydrologic and agent-based salamander model is also created. The goal of the project is to deduce salamander populations from climate information.

The results of the modeling exercise show that automated model integration was enabled by an ontology that combined semantic and procedural knowledge. Further, the results demonstrate that immutable and partially mutable data properties provide the uniqueness property necessary for the reasoning engine to identify data set relationships and dependencies. With the metamodel and data semantic and procedural knowledge models the reasoning engine is able to integrate disparate computational models into a powerful deductive logic workflow engine.

4.1 Introduction

Model integration is becoming increasingly used and an important tool to examine and engineer multiple systems. Increasingly water managers are being asked to examine the impacts of water management options on water quality and ecology. Model integration requires knowledge of data sets, such as the type of data, whether the data is historic data or model-produced data for a scenario, time and space scale (internal scale and extent), and the units. Also included in this body of knowledge are the procedures for creating the workflows and transforming one data set into another. Semantic modeling can facilitate model integration. One of the goals of semantic modeling is to facilitate automated analyses of

complex situations. Computational models, such as hydrologic models, are complex tools that involve many data sets. They often involve many parameters and describe many different physical processes. The task of analyzing and transforming significant amounts of data into the physical process parameters, as well as the task of transforming the data into the description of the geometry of the problem for the numerical algorithms, is not simple and straightforward. Often the results of one analysis can dictate other analysis steps. As such they are not easily added to workflows. They also involve a wide range of concepts, from the physical processes they model (including appropriate scales), to describing and creating appropriate geometry for the numerical model, including boundary conditions and model parameterization. Thus the semantic knowledge set required to fully describe computational models is significantly large and costs a significant amount of effort to put in to standard semantic models. Yet these concepts represent a significant body of knowledge of process representation.

It is the interaction of physical processes, many of which are represented in computational models, which are key to understanding the overall system impacts to management options. While it is often desirable to examine, for example, ecological impacts of watershed management decisions, often smaller projects just do not have the budget for creating and integrating ecological models on top of hydrologic models. It is practical limitations, rather than theoretical limitations, that limit systems models. If these computational models could be constructed and connected in an automated fashion, though, then there would be little in the way of examining a wider set of consequences for decisions.

The goal of this paper is to demonstrate that concepts about disparate computational models can be incorporated into a knowledge base that allows a semantic reasoning engine to reason about and execute them. This is done using a new integrated semantic and procedural knowledge reasoning engine and knowledge base that we have developed (see Chapter 2). The computational models are automatically included in a workflow as part of the deductive process, creating a set of loosely-coupled models. The key to the seamless integration of computational models into the deductive workflow is the use of a semantic metamodel of computational models. Each computational model is unique, and as such semantic models describing them will be unique. But the overall purpose of a computational model used in the physical sciences is to transform one set of data into another. Generalizing this knowledge about computational models creates a metamodel that the reasoning engine is able to use to reason about the different models.

Computational models are created to be used, in general, in a decision making processes. The key driver of the decision process, though, is not the models but rather the data. The computational models are the tools for deducing the desired data from other data. When viewed as such, computational models would seem a natural fit with reasoning engines, albeit they operate on very complex sets of knowledge. Reasoning over data and the relationships between data sets is where the semantic and procedural reasoning engine can be of especial assistance. For this purpose a suite of ontologies have been created that describe what it means to be a data set as well as time and space scaling and scale comparison of data sets. By being precise about the data requirements the reasoning engine can use the scale information to find relationships between desired data sets and data sets output by computational model as well as historical data sets. The key data set properties

are time and space extent and internal scale, data type, and the scenario it is related to (including the historic scenario for measured data).

To demonstrate the viability of this semantic and procedural approach to model integration a logical infrastructure will be created that a) describe the computation models,2) describe classes of simulations for particular computational engines, and 3) describe project frameworks. Two computational modeling engines are used, the first is a stochastic weather model written in the R scripting language (R, 2011) and the second is a hydrologic and ecologic agent-based model written as a NetLogo (Wilensky, 1999) model. NetLogo is a program to create and run agent-based models. NetLogo, and thus the NetLogo simulations, run on Java (Oracle, 2011). The data creation project, which ends up using these two simulations, examines the effect of climate change projections on the population viability of an ecologically sensitive species, the Frosted Flatwoods Salamander. The salamanders depend on specific hydrologic situations in order reproduce and the climate change projections influence the hydrology of the modeled area.

4.1.1 Background

Semantic modeling is the modeling of individual concepts and their relationships to other concepts. The relationships between the concepts form a web or graph. The fundamental idea behind semantic modeling is that it is the relationships between the concepts that give the concepts meaning. For example, a set of numbers takes on meaning when they are expressed as representing measurements of some physical property. In hydrology we often use informal semantic models to represent the many concepts we work with and how they relate to the tools and computational models we use. Formal semantic

models are based on globally unique representations of concepts and formal semantic logics called descriptive logics.

Semantic models are often recorded using XML. One of the advantages to using XML is that the URL can be set up as a namespace and the actual unique concept name shortened for increased human readability without sacrificing the full URL representation of the concept.

4.1.2 Functional Ontologies

Functional ontology (see Chapter 2) logic is a formally defined semantic logic (http://chl.erdc.usace.army.mil/FO-20111201#) that extends the concepts of semantic modeling by providing for specific properties of conceptual relationships. These properties of the relationships are functional aspects of the relationship concept. The power of the functional ontologies is what enables the creation of the overarching logical infrastructure.

4.1.3 Metamodels

The term "metamodel" has been coined to refer to models of models. Metamodels abstract and generalize features of the underlying models. Metamodels are used in two manners. The first is the abstraction of software and the second is in semantic models. The Model Driven Software Engineering (MDSE) community uses metamodels (and meta-metamodels, or megamodels) such as Unified Modeling Language (UML, 2010) diagrams to represent underlying software concepts and relationships. The goal of these metamodels is to enable the automated conversion of one software description into another or the automated creation of software from the description diagram.

A recent advance in business process modeling also includes metamodels and illustrates the use of metamodels for semantic modeling. ConceptBase (Jeusfeld, 2009) includes the concept of metamodels, in the sense of models of semantic models used for business database processes (Jarke et al., 2009). The goal of these metamodels is to enable the integration of legacy database process models into a more complex system.

Semantically defined web services (Payne and Lassila, 2004) have also benefitted from recent work in metamodeling. Semantic web services are web services whose interface is defined in the usual (e.g. REST or SOAP) form but is also described by a semantic model. Much like the ConceptBase metamodels, the Semantic Web Service metamodels are used to abstract the properties of unique and varied semantic web services in order to facilitate their integration into a larger system of services (De Virgilio, 2010).

Given the success in using metamodels to integrate disparate and varied systems, this work will investigate their use as part of a functional ontology in order to reason about using disparate and varied computational models to deduce new knowledge from existing knowledge.

4.2 Methods

In order to investigate the integration of computational models into a semantic and procedural knowledge framework, a logical infrastructure will be created and tested. This logical infrastructure will include a range of metamodels for data sets, computational models, and projects which control the structure of the data as well as provide critical functionality. Finally, these models will be used in an example that examines deducing the effects of climate change on the viability of an endangered ecological species.

4.2.1 Semantic Metamodels of Computational Models

Metamodels are semantic models that describe a class of semantic models, similar to a schema or template. By creating metamodels and utilizing them to describe the actual semantic models, we are able to express in general terms properties and attributes of a whole class of semantic models. This approach greatly increases our ability to incorporate specific semantic models into a more general semantic modeling framework.

As a directly relevant example, consider computational models. Conceptually, computational models embody a functionality designed to compute one set of information from another. So while we could make semantic models of specific computational models, their use would be applicable only to other semantic models directly related to the first. For example, Figure 4-11(a) shows two simple semantic models about computational models called “Computational Model X” and ”Computational Model Y.” If there was a need for a workflow engine to compute the data set D:X_Data then we could use “Computational Model X.” However, for the workflow engine to make use of this knowledge it needs an inherent understanding of the meaning of X:hasOutputDataSet. If the workflow engine also needs to compute the data set D:Y_Data then it can use “Computational Model Y” but, again, only if the workflow engine has an inherent understanding of the meaning of the concept Y:ComputesDataSet. This requirement imposes a significant burden on the workflow engine and quickly becomes untenable as more and more models are added to the list of available computational models.

What is needed is for the workflow engine to have to understand the meaning of X:hasInputDataSet, X:hasOutputDataSet, Y:ComputesDataSet, and Y:NeedsDataSet apart from their relationships shown in the diagram. What is missing is the knowledge that

X:hasOutputDataSet and Y:ComputesDataSet are both predicates denoting model output. Similarly X:hasInputDataSet and Y:NeedsDataSet are both predicates denoting input data requirements. Enter the metamodel shown in Figure 4-11(b). The metamodel "Computational Model Class" is a semantic model of semantic models, specifically semantic models of computational models. It defines computational models as having input and output predicates. Below the "Computational Model Class" are the definition of the relationships for "Computational Model X" and "Computational Model Y." Given this new information, a workflow engine would only need to know about the concepts of the metamodel in order to deduce the correct workflow. Whether one or one hundred models are part of the model knowledge base, the workflow engine can just as easily search and deduce the workflow. Thus the metamodeling approach gives the workflow engine unlimited flexibility to incorporate computational models.

Given that model integration involves more than one model, a computational model metamodel has been created as part of this work effort. The metamodel is somewhat more complicated than the simple one shown in Figure 4-11. In creating the metamodel, there are really two parts, creating concept classes to represent the abstract ideas, and defining predicates classes that represent the abstract relationships. Figure 4-12 shows a more complete conceptual metamodel of computational models as well as the symbol used for computational models in the abstract schematic diagrams included later in Figure 4-17.

The computational model semantic metamodel created for this work actually has a slightly different view of computational modeling, one that better facilitates a collection of computational model instances. The metamodel defines what is termed a "Model Class." Model classes are viewed as describing general properties of a computational model engine,

such as general input and output data sets as well as other typical *attributes* of models (not properties, although they could. See also Table 4-1.) These attributes could be, for example, that the computational model uses a defined time step or uses one of a set of numerical integration methods. These attributes of the model class are used by the model class itself, but not, in general, outside of the model class. Model classes include references to what are termed "reference models," which are specific instances of computational models and simulation data that have been previously set up. It is these reference models which are able to actually compute data sets; the model class is an abstraction level of the group of reference models. The computational model semantic metamodel defines relationships for both model classes and reference models.

4.2.2 Semantic Metamodeling and Procedural Knowledge

When considering metamodels as semantic models in and of themselves, if is fairly straightforward to identify functionality that applies to the metamodels and thus to the semantic models as well. For the computational model metamodel example, computing a desired data set is an obvious example. A somewhat more subtle example is computing the dependency chain for a given data set. For a semantic model of a computational model, the dependency of an output data set on the input data sets is a second-order logic relationship (e.g. there are two dependency links between data sets, meaning that the output data is created by the computational model, and the computational model required input data). Adding the computational model semantic metamodel to the mix and the dependency becomes a highly complex third-order logic (depends on the connections to both the metamodel and the computational model) where the logic connections are independently

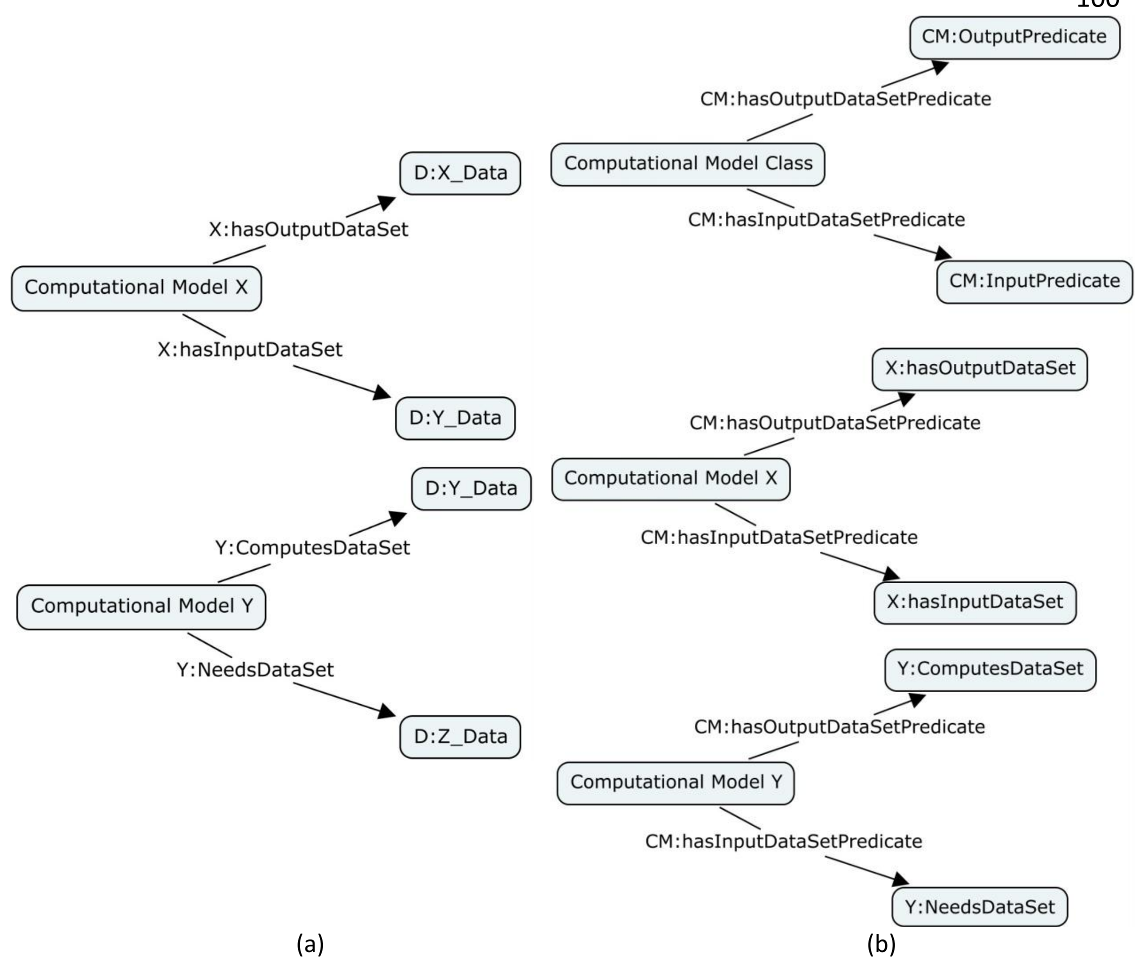


Figure 4-11. Simple semantic model and metamodel of computational models. Part a shows a semantic model for "Computational Model X" and "Computational Model Y," while part b shows the metamodel "Computational Model Class" and how to describe "Computational Model X" and "Computational Model Y" in terms of the metamodel.

defined (but conform to the metamodel description), not something easily deduced by a reasoning engine based on first and second-order logics.

Adding the data set format described below increases the dependency of one type of data on another two steps to a fifth-order logic – data types are part of data sets (one step), data sets are computed by computational models (one step) that consume input data sets (one step) that have data (one step), the description of computational models are independent but defined to conform to a metamodel (one step). A seemingly simple

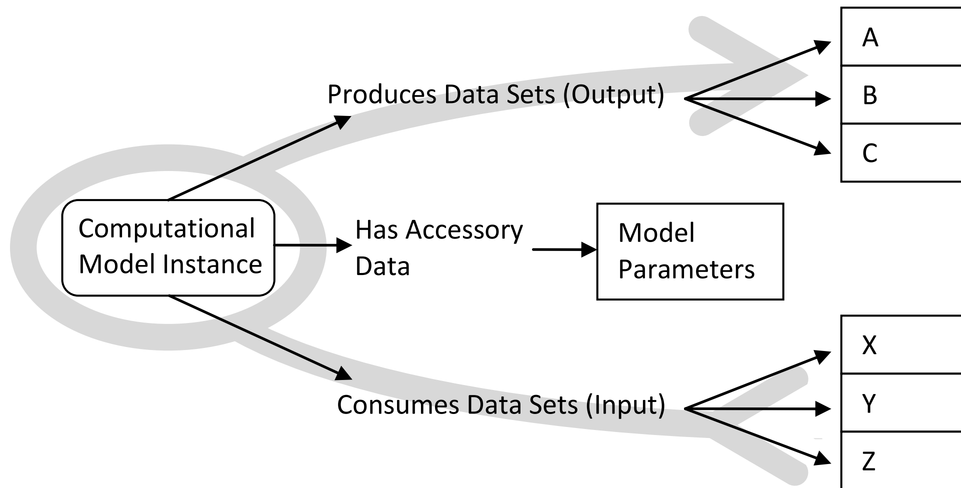


Figure 4-12. "Meta" conceptual model of computational models. In this form the relationship between the output data sets and the input data sets is a second-order logic relationship with the computational model forming the vital relationship link.

question as “What data do I need to compute a desired result or output” is not so simple after all. It is, however, quite amenable to being described as an algorithm that can be included as a procedural aspect of a predicate. Thus, a useful metamodel predicate would be a predicate for the dependency of one data on another.

With the computational model semantic metamodel, a dependency predicate, and a “create and execute the model” predicate the functional ontology reasoning engine actually becomes a powerful workflow engine in and of itself, not only able to execute computational models but integrate a broad spectrum of metamodels with an inherent understanding of the meaning of data. In essence the semantic and procedural aspects of the computational model metamodel represent, in a straightforward manner, the abilities of a workflow engine. By enabling procedural aspects of knowledge to be represented in a directly usable fashion the functional ontology approach to knowledge modeling creates a powerful mechanism to deduce, create, and transform knowledge.

Table 4-1. Computational model metamodel predicate definitions.

Predicate	Meaning
md:hasReferenceModelPredicate	A semantic model of computational models include instances of simulations for the modeling engine. Reference models are these simulation instances that can be customized to produce data sets for specific scenarios. The semantic model must have a predicate to relate these reference models to the model class definition.
md:hasReadModelConcept	Defines the predicate used by the semantic model for reading in a reference simulation.
md:hasCreateModelConcept	Defines the predicate used by the semantic model for creating and running an instance of a reference model customized for a scenario. The predicate function should also write the output data sets in the format needed by the computational model and read in the resulting data sets.
md:hasOutputDataPredicate	Defines the predicate that used to identify output data created by a model class.
md:hasInputDataPredicate	Defines the predicate that used to identify input data created by a model class.
md:hasReferenceModelOutputDataSetPredicate	Defines the predicate used to define actual data sets set up as output data sets of the reference simulations. These data sets have time and space scales and data types but do not have scenario specifications or actual data values.
md:hasReferenceModelInputDataSetPredicate	Defines the predicate used to define actual data sets set up as input data sets of the reference simulations. These data sets have time and space scales and data types but do not have scenario specifications or actual data values.

4.2.3 Semantic Models of Data Sets

Computational models, whether physics-based numerical models, analytical models, statistical models, or otherwise, are designed to represent a process that transforms one type of data, one type of knowledge, into another. The data for computational model used in the physical sciences is frequently of a physical-world, geosciences context. It is this type of data that will be considered for this work.

As engineers and scientists we use computational models to assist us in analyzing data and to produce results, or output data upon which we make decisions. For example, if a runoff model tells us that a new proposed subdivision will result in increased flooding downstream we require mitigation measures of the developer. If a tsunami model tells us a tsunami is on the way we alert the populations at risk so they can evacuate. These decisions are based on the data produced by models.

In computational modeling there are two broad classes of knowledge that must be considered and weighed in order to adequately provide new knowledge: knowledge about the models and knowledge about the data. Computational models encode a significant amount of knowledge about physical processes and logical data transformation pathways. In general, since formal semantic modeling has only recently started to be used in conjunction with computational modeling, computational models are built on informal (and often implicit) semantic models. Formal semantic models of the computational models thus help to bridge the gap between the informal semantics of the computational model and a formal semantic representation that facilitates integration with other semantic models, including data. Computational models, though, are just a means to capture and use formal procedural knowledge in order to transform one knowledge form into another; they represent, in a formal procedural manner, the relationship between data. It is not the exercise of the procedural knowledge that is desired rather the end results of the execution. The computational models are thus just a tool to deduce mathematically, results in the form of output data based on the knowledge represented by their encoding. Computational models, then, can be viewed as a source or store of knowledge. It is this knowledge that must be used

for real-world decisions, not the models. Thus a data-centric view of model integration is both needful and appropriate.

When looking at how models use data, and how we set up computational modeling projects to answer questions, the various dimensions of data become apparent. These are illustrated in Figure 4-13 and Figure 4-14. The first dimension is the fundamental meaning of the data; these are not merely meaningless values but have some real-world physical meaning, such as air temperature, stream stage, or population count. A second dimension involves the representation of the data – the data we measure or model is only a subset of all the possible data. It has both spatial and temporal extents as well as internal scaling. The third dimension of data comes to bear from looking at modeling projects. Often computational modeling projects are designed to look at "what-if" scenarios – what if the city picks land use plan A over plan B, what if a levee is put in along the river, what if an approaching hurricane changes it course, what if carbon emission scenario A1B comes to pass, etc. Computational models are designed to help answer these questions, but it should be recognized that an integral part of a data model is the alternative or project forcing scenario used to produce the data. It should also be noted that what differentiates scenarios are the data sets designed as alternate versions of what could be.

4.2.4 Deductive Logic for Data Sets

To enable the deductive logic for the data sets, the procedural knowledge examining and evaluating the relationships between data sets will be encoded as part of an ontology of data sets. To determine if one data set can be created from another (e.g. the input data set for one computational model from the output data set for another computational model) the properties of each data set are compared. The properties of datasets are the key to uniquely

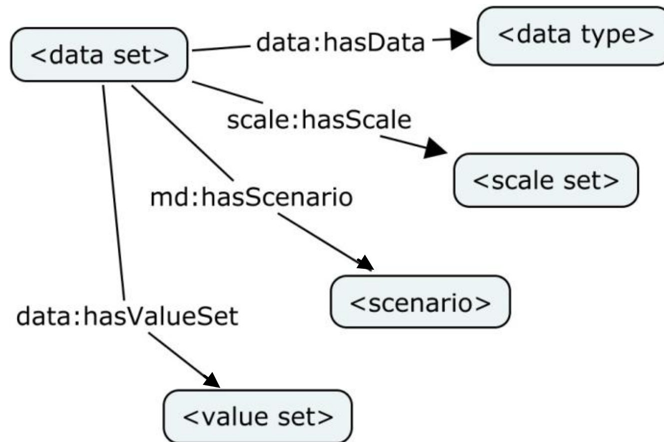


Figure 4-13. Semantic model of data sets includes several attributes that act as dimensions to the data set.

identifying them for use as part of the reasoning process. There are three types of properties: immutable (can't change) properties, such as the data type and the scenario used to create the data set, fully mutable (can change, fully reversible) properties, such as the units, and partially mutable (can change but not in a reversible sense) properties, such as being able to scale from a fine scale to a coarse scale.

The immutable properties are used to uniquely identify one data set from another. The partially mutable properties are used to clarify whether one can be turned into another. The mutable properties simply clarify the form of the data. By testing the immutable and partially mutable properties the reasoning engine will be able to identify which data sets directly relate to others in a mutable-only fashion. Data sets that are immutable must be deduced by the computational models.

The first two data tests are on the immutable properties – matching the data type, scenario, and extent. Once the reasoning engine verifies that they match then the internal scale information is checked. The scale information is built on a "scale" ontology that defines

the properties of a scale set. The scaling ontology, meant to be a proof-of-concept ontology, is shown in Figure 4-15. The scale includes time and space domains, both extent and internal scale. The time and space extent triples are actually based on other ontologies, ontologies of space and time. The space ontology (describes where things are at) is an extension of the GeoRSS simple model (GeoRSS, 2012). The GeoRSS simple model defines simple feature objects such as points, lines, and polygons. The functional ontology space ontology adds the concepts of an envelope (feature extended by a radius) and whether or not one envelope encompasses another. Procedures are used to compute the envelope (fgeo:Envelope) and whether one envelope covers (fgeo:FullyEnvelopes) another. The time ontology, while simpler, has similar functionality. The scale ontology uses this functionality to see if one data set is fully enveloped by another, both spatially and temporally.

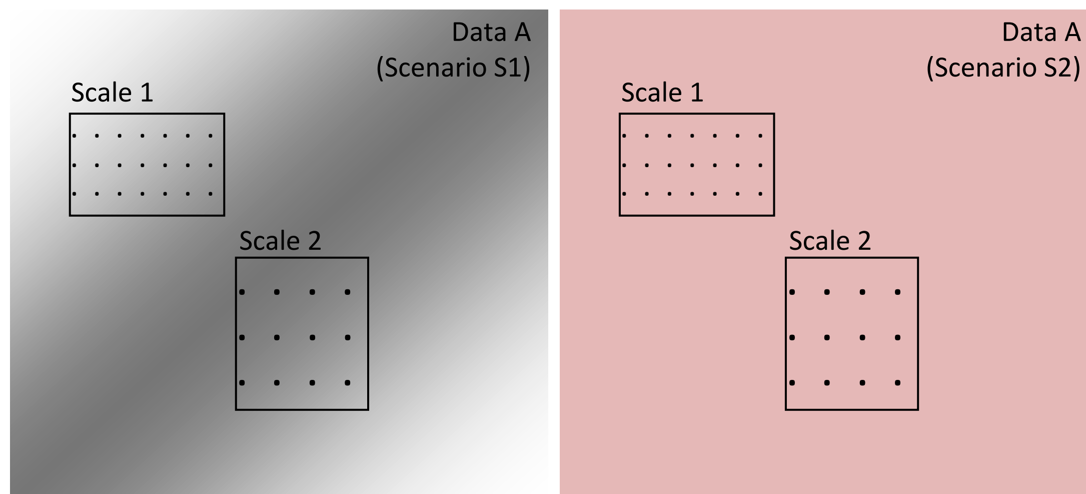


Figure 4-14. Representation of the relationships between data, scale, and scenario for a data set. A data set consists of values from a data set for a particular scenario. The extent and internal spacing of values is determined by the scaling data. “Data” is considered a spatially and temporally continuous property. Data sets are windows into “data” values.

After it is determined that one data set extent covers the other, the internal scale is then checked to see if the data can be transformed from one to the other. The order, for both time and space, of internal scale types is that point values can be derived from real-valued scales, and real-valued scales can be derived from other real-valued scales or uniform scales. Point scales refer to the data set being a single value, either in space or time. Real-valued means that the data values are continuous over the interval but broken up into unit parts, which are of the size <internal scale value>. For example, hourly time series for an airport meteorologic data would have internal space scale <scale:PointValue> and internal time scale <scale:RealValued> with internal time value "0.041667," which is one hour expressed in Julian time (one day = 1.0). The scale ontology has procedures for determining if one internal scale is equal to or can be upscaled to another. Downscaling is not permitted at the moment.

4.2.5 Procedural Modeling for Data Values

There are a large number of data types, from single values to multi-dimensional time series. The goal of the ontology is to describe the data sufficient that the reasoning engine can use the data. The underlying data storage mechanism, however, is open to different forms. For example, the parameter and statistical data is stored as a set of triples in the ontology whereas time series data is stored in a class container in the common code of the ontology. Another data type created is a two-dimensional gridded time series where the time data is stored similar to the time series but the data is cached on the hard drive until needed. So long as the procedural knowledge is able to answer the queries presented the underlying form is unimportant. Since data transformations are an important part of the model integration process, the procedural knowledge is expected to provide the interpolation and

transformation tools, either using strict procedural knowledge or a combination of semantic and procedural knowledge.

4.2.6 Modular Ontological Engineering

The development of the semantic models involved creating different sets of closely related concepts into individual ontologies. This approach allows the individual ontologies to be developed in tandem. There are several different flavors of ontologies that were developed and used. The "Upper Level" ontologies are used to define the data types, such as "count:Population" or "thic:Precipitation." Using these ontologies enables disparate models and data sets to be identified in uniform terms. The application-level ontologies (Table 4-2) form the foundational semantics and capabilities. These include the scale ontology, the time series ontology, etc. The metamodel ontologies (Table 4-3) utilize, integrate, and define semantics about the application-level ontologies. These include the Computational Model semantic metamodel, the data set metamodel, and one for reading card-based project files of computational models. The conceptual relationships between the ontologies are shown in Figure 4-16.

4.2.7 Pulling It All Together: Determining Model Integration Requirements

The computational model metamodel, and thus the computational models, is framed in terms of transforming inputs into outputs. To create desired outputs requires specific inputs. Thus the driving force for model integration are the data requirements for each computational model step. The project metamodel defines the desired data set to be used for decisions – this defines the first step of the analysis process. The task of the reasoning engine is to use the logical infrastructure as well as the actual model descriptions to deduce

the path from the scenario data sets to the decision data set. The process is illustrated in Figure 4-17. The complete knowledge base includes a collection of reference models (actual simulations), actual complete data sets (either historic or for a scenario) such as the scenario input data sets, and the project specification. The reference models refer to both input and output data sets that have a specified data type (the hollow shapes) and scale (internal and extent). For example, the M1 model class has a reference model that takes a “star” type data set with scale S1.5 and transforms it into a “triangle” data set with scale S1.7. The “1” in the scale refers to an extent while the “.5” and “.7” refer to internal scales.

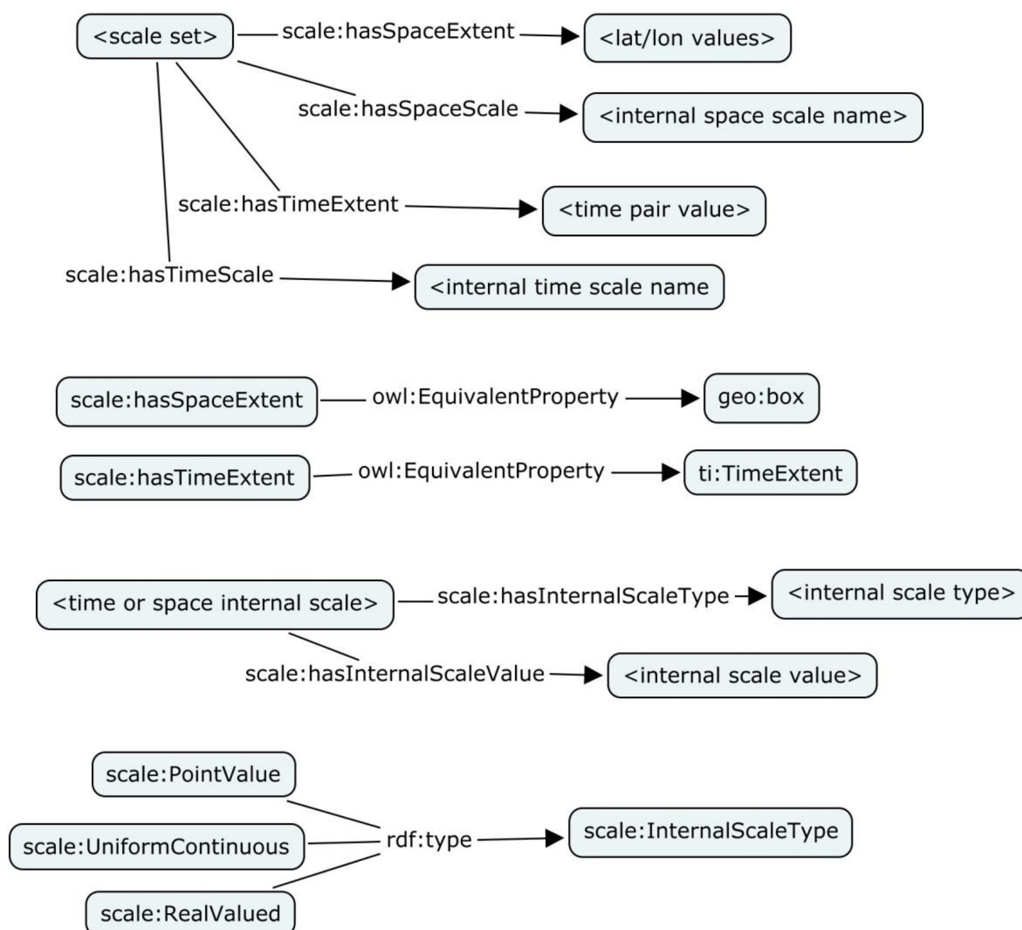


Figure 4-15. An ontology of scale. This includes the concepts of extent and internal scale, both for time and space.

Table 4-2. Descriptions of the application-level ontologies developed for this project.

Application-level Ontology	**Semantic Knowledge Description**	**Procedural Knowledge Description**
Time	Julian and Gregorian time	Convert between Julian and Gregorian time formats
Time Series (single)	Defines concepts relating to a time series of values	Interpolation of values between given points; integration of data from one time to another
Scale	Defines extent and internal space and time scales for data sets	Check to see if one scale set is enveloped by another as well as if one internal scale is more refined than another (for upscaling)
Functional Geography (space)	Builds on the OWL Geo ontology; defines envelope	Computes envelope of a geographic entity; computes if one entity envelopes another entity spatially
Parameter	Defines a real/integer/string parameter and its association with a parameter identification	Read from text file
Statistics	Defines a pair of second-order statistics (mean, standard deviation) as well as a group of monthly second order statistics	Add/query for monthly statistics based on month number rather than just name.
Command-line execution	Defines concepts around executing a command-line function	Execute a system call to execute a command-line call.

The deduction logic works backwards from the desired data set to the input scenario data sets. When the reasoning engine is asked to create the “diamond” S2.9 historic data set, shown in the project specifications panel of Figure 4-17, it searches for a data set and deduces that it can use the “diamond” S2.4 data set. It does this by looking examining all the data sets (and model produced data sets) of the correct type to find one at the correct location and space scale. Since the data set does not exist it queries for the reference model

Table 4-3. Description of the metamodels created for the project.

Metamodel	**Semantic Knowledge Description**	**Procedural Knowledge Description**
Data Set	Defines a data set as having a data type, set of data values, and a set of scale information (extent, internal scale). Also defines stochastic data sets and the “historic” or measured data scenario for use by all measured data.	Checks if a desired scenario data set already exists; computes stochastic data set count
Model Project Alternative Scenario	Defines the relationship between projects, project decision data sets, alternative scenarios and scenario data sets, computational model classes and the conceptual meta-model for computational models.	Find or compute project decision data sets and scenario data sets; read model, execute model “meta” procedures
Card-based Project	Defines concepts that must be specified by models that have card-based project files to automate reading and writing	Reading and writing the file, setting up the parameters

used to create the data set and executes the metamodel’s CreateModel query (part of the logical infrastructure to be able to execute the models).

The CreateModel query of the metamodel checks to make sure the input data sets are satisfied for the reference model, queries for the actual CreateModel predicate for the reference model’s model class, and then calls that CreateModel query. If the data sets (e.g. historic “star” 2.4) are not satisfied the CreateModel predicate of the metamodel begins the search anew for the input data sets. In this manner the reasoning engine steps backwards, deducing the previous link is the data transformation chain. Once it reaches to the existing data set for the scenario all the input data sets are satisfied and the computations begin and proceed up the modeling chain. If at any time a required data set does not exist, and the reasoning engine cannot find a data set that can be used to create it, the chain stops and the

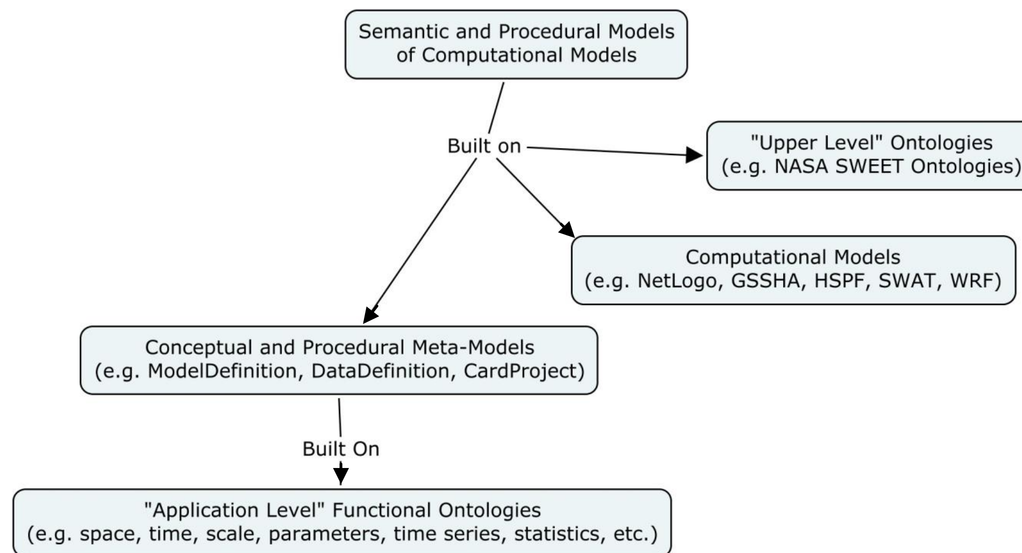


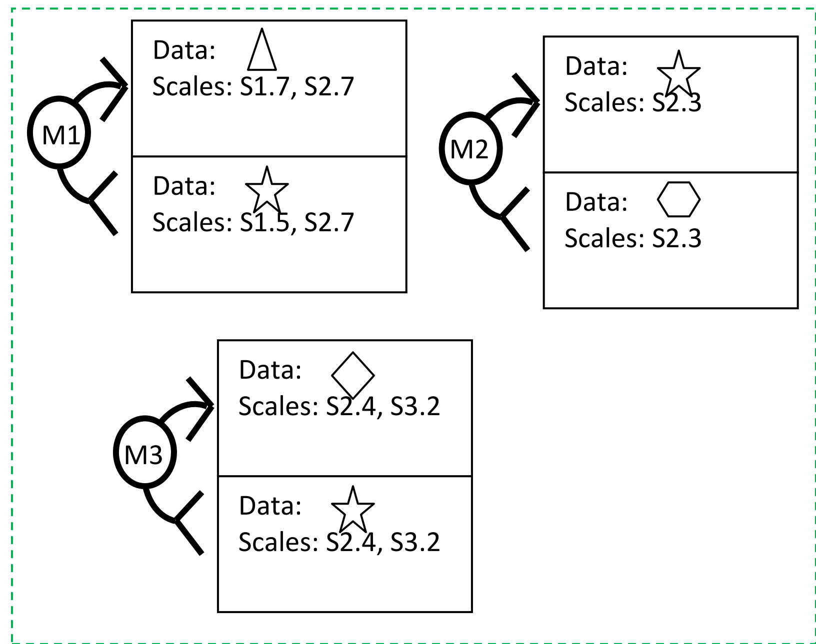
Figure 4-16. Relationship between the several ontologies used to create the semantic and procedural models of conceptual models.

reasoning engine reports the missing data sets. The “deduced workflow” panel also shows how data sets computed for a scenario for one project can be used for the same scenario but for a different project – the scenario is the key data set dimension, not the project it was created for.

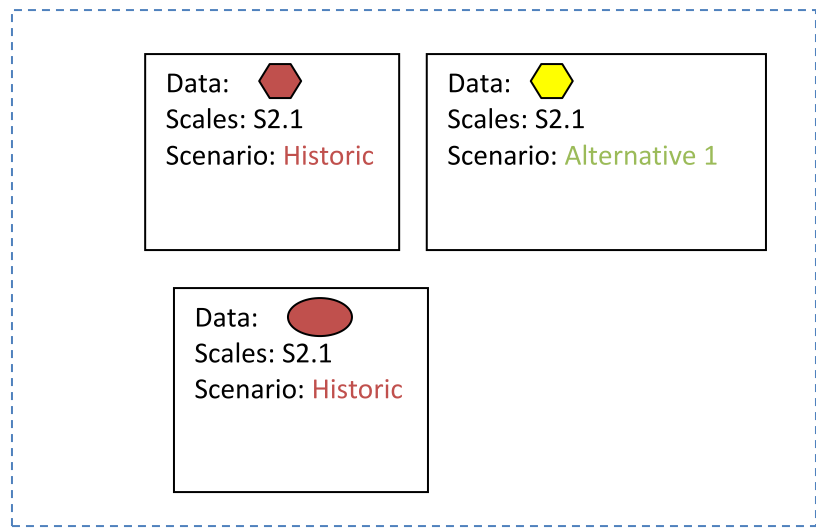
4.3 Demonstration Model Integration Project

To demonstrate the utility of this semantic and procedural model and metamodeling approach, a proof-of-concept model integration project will be conducted that examines the impact of climate change on a hydrology-sensitive endangered species, the Flatwoods Salamander. This proof-of-concept is actually the first step in a more comprehensive analysis of Ft. Stewart, GA. This project will be completed by June 2013 and will be published by September 2013. Fort Stewart in Georgia, USA, is used to train military personnel.

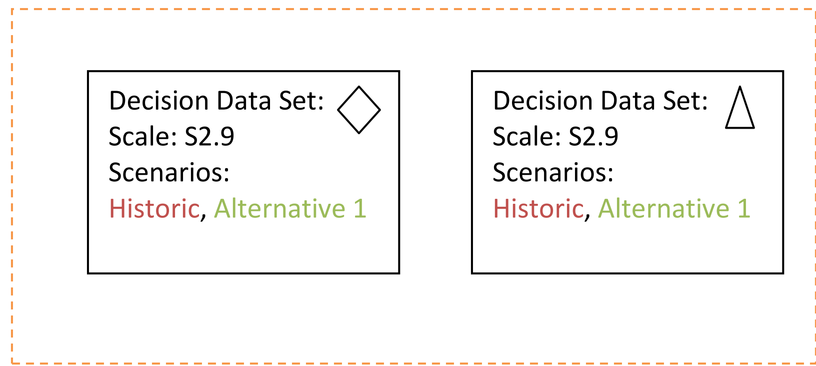
Model Collection



Data Set Collection



Project Specifications



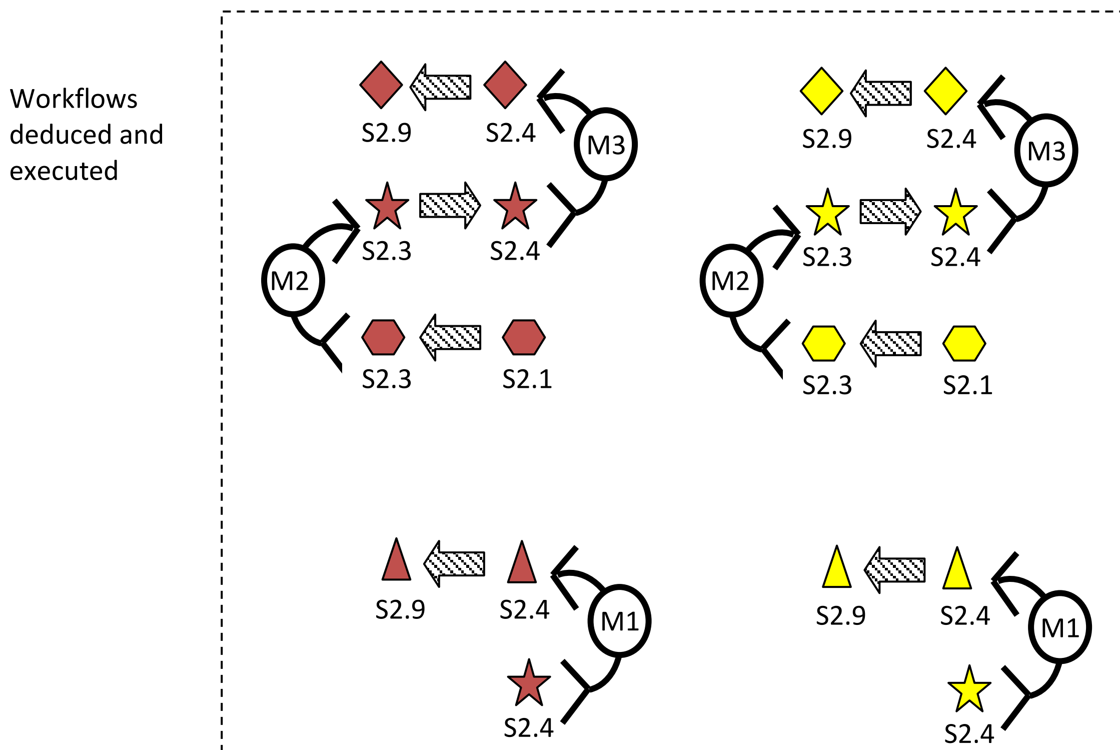


Figure 4-17. A simple example showing a collection of models, existing data sets, project descriptions, and the final workflows deduced and executed. The model collection represents models as circles (e.g. M1, M2, etc.) with the curvy arrows as input and output data types. The various data types are represented as hollow shapes (star, diamond, hexagon, etc.). The data set collections shows data values for a data type, with a scale and scenario. The filled-in shapes represent data sets with the color (e.g. red, yellow) being the scenario. The S numbers below the shapes represent the scales, both extent and internal. The first number refers to an extent, so S1 would be a different location than S2, and the second number refers to the internal scale, with smaller number being more refined than larger numbers.

The Flatwoods Salamander is found around a few ephemeral ponds. While the Flatwoods Salamander is both present and classified as endangered or threatened, training is restricted from those areas. The population level of the Flatwoods Salamander is thus a key variable of concern and used for decisions about the training facility. To model the Flatwoods

Salamander an agent-based simulation has been created using the NetLogo agent-based modeling software (Wilensky, 1999).

In the agent-based salamander model, the physical environment in which the salamanders live is simulated as well as “agents” that are programmed to act as the salamanders. The environment includes elevation, surface water, vegetation, soil, and weather properties. The surface water, soil moisture, and weather properties vary over time. The salamander agents decide where the salamander should live, where it should move, and if it is able to reproduce or not. The salamander agents grow through the life cycle of the salamander and, if the environmental conditions are not sufficient or the agent is older than the general maximum age of salamanders, the agent dies. For more information see Westervelt et al. (2012).

Because the key driver of the population lifecycle of the salamander is the presence of the nearby ephemeral ponds, a simple pond filling and depletion hydrologic model has been created as part of the environmental simulation. The key inputs for the hydrologic portion of the model are temperature and precipitation. The output of the simulation is salamander population level over time. The key scenario variables of interest are climatic variations from any one of a number of climate change scenarios. To bridge the gap from climate variations to weather, two models are used. The first is a Weather Research and Forecasting Model (WRF) (Michalakes et al., 2004) simulation run on a supercomputer. This model takes Global Climate Model (GCM) inputs as boundary conditions and outputs regional weather data, which is then analyzed and turned into climate change statistics.

The next model is a stochastic weather model that takes the climate change statistics and stochastically creates sets of weather data. For the full modeling scenario another model

will be next in line, a hydrologic model that takes each of the weather sets and simulates the physical processes resulting in surface water ponding. For this example, however, the simpler hydrologic model included in the salamander agent model will be used. When completed, the salamander model will take both weather data and ponded depth information. While the WRF model is run on supercomputers, the stochastic weather model, the detailed hydrology model, and the salamander agent-based model can all be run on personal computers. This proof-of-concept will begin with the output of the WRF model as the scenario data and the project decision data will be the salamander population levels for a specified pond. The complete project will also incorporate a risk model that examines the probability of each of the scenarios and the expected outcome of the population under the scenario and forecasts expected risk for the installation.

4.3.1 Modeling Domain

The modeling area, shown in Figure 4-18, is a set of small ephemeral ponds on Fort Stewart, GA. However, the different reference models have various domains. The stochastic weather model is for the Savannah/Hilton Head Airport, in Savannah, GA. The WRF model covers the southeastern portion of the U.S. Table 4-4 shows the input and output data for each of the various models.

Table 4-4. Each of the computational models for the complete project requires different input and output data sets. The computational models act as the relationship links between the data sets. Two models will be used for the model integration proof-of-concept demonstration, the synthetic weather model and the Flatwoods Salamander model. Preliminary data from the WRF model will be used as an alternative scenario input data.

Computational Model	**Input Data**	**Output Data**
Global Climate Model (GCM)	Climate Scenarios	Global weather data
Weather Research and Forecasting (WRF) (and post-analysis)	Global weather data at region boundaries	From WRF: regional weather data From post-analysis: Climate statistics
Synthetic weather model	Climate Statistics	Set of time series weather data for a location (stochastic representation of climate)
Gridded Surface Subsurface Hydrologic Analysis (GSSHA) hydrologic model	Precipitation, meteorological data, digital elevation data, soils data, land use data, stream profiles	Surface water depths (in depressional ponds)
NetLogo Flatwoods Salamander Model	Surface water depth, meteorological data	Flatwoods Salamander population count
Risk Model	Flatwoods Salamander population count	Risk to facility

4.3.2 WRF Scenario Input Data

Two climate scenarios will be run for the proof-of-concept modeling exercise, a historic scenario and a climate change scenario loosely based on the Inter-governmental Panel for Climate Change's Special Report Emissions Scenario (Nakićenović and Swart, 2000) A1B scenario.

Figure 4-19 shows the two sets of precipitation statistics while Figure 4-20 shows the two sets of temperature statistics. These statistics are used to drive differences in the stochastic sets of weather data. Table 4-5 shows the scale information used with the WRF data.

4.3.3 Stochastic Weather Modeling

The stochastic weather model, created to reproduce historical statistics, including auto- and cross-correlations, for eight hydrometeorological data at an hourly scale: precipitation, temperature, wind speed, barometric pressure, relative humidity, cloud cover, and direct and total radiation. The weather model is a stochastic time series model that will create hourly data for all eight variables, including cross-correlations. The weather model

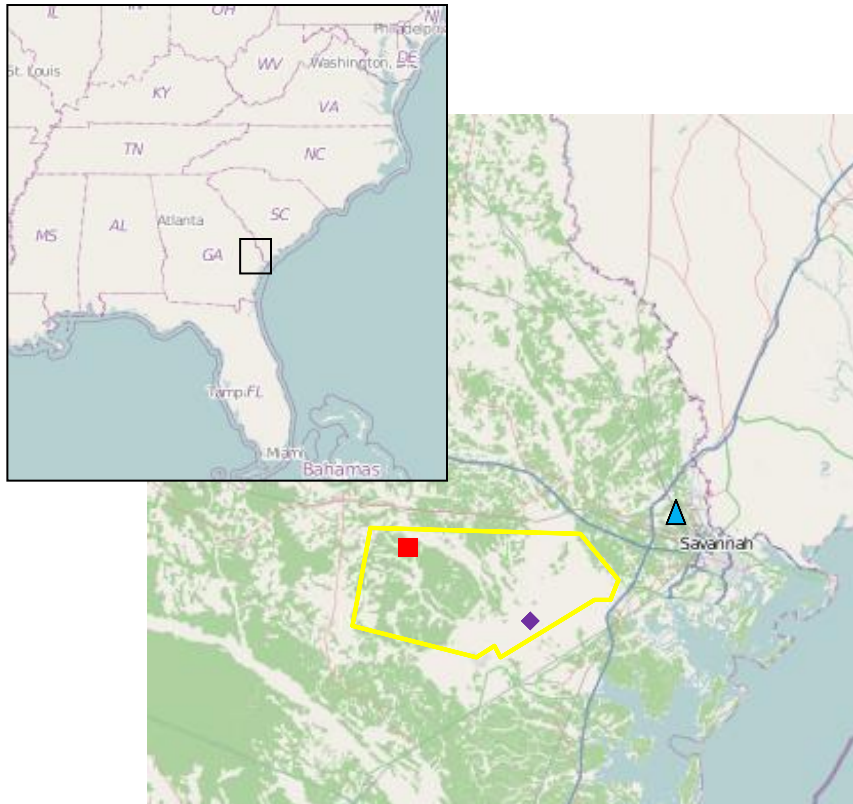


Figure 4-18. Locations of the ecological model domain (red square) and the Savannah/Hilton Head Airport (blue triangle). The yellow outline is approximately the boundary of Fort Stewart. The purple diamond is the approximate location of the pond shown in Figure 4-21(a) where salamanders were most recently sighted. Background map imagery © OpenStreetMap contributors, CC BY-SA[5].

[5] The Creative Commons Attribute-ShareAlike 2.0- license (CC BY-SA) permits copying of the imagery provided the attribution statement is included.

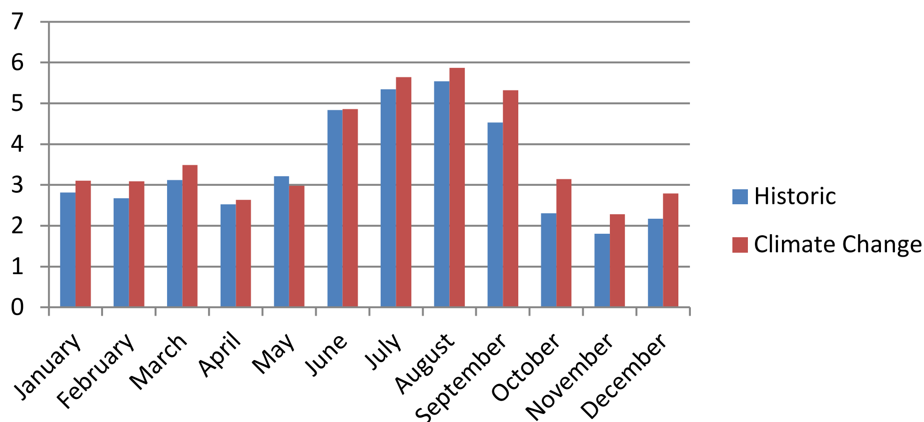


Figure 4-19. Precipitation statistics used for the two climate scenarios.

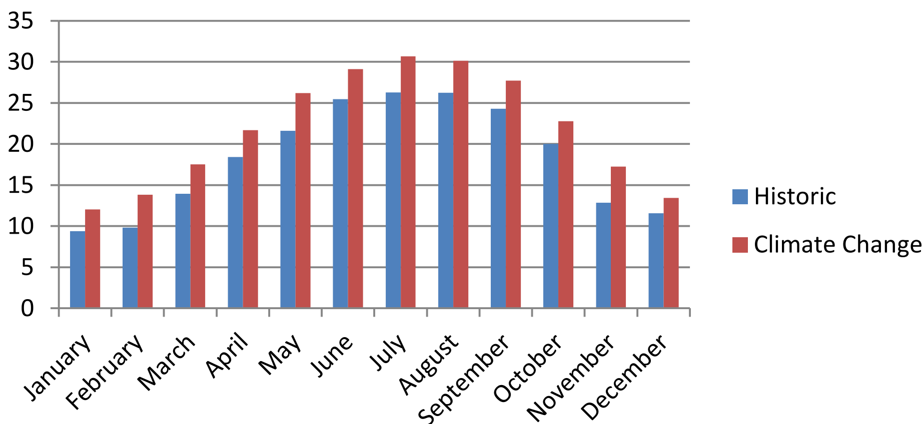


Figure 4-20. Temperature statistics for the two climate scenarios.

Table 4-5. Scale triples for the WRF data used as the input scenario data. The times are Julian values.

cdr:WRFRegionExtent	scale:hasSpaceExtent	37.391027 -90.765781 29.465569 -75.438652
cdr:WRFRegionExtent	scale:hasSpaceScale	cdr:WRFRegion_SpaceScale
cdr:WRFRegion_SpaceScale	scale:hasInternalScaleType	scale:UniformContinuous
cdr:WRFRegionExtent	scale:hasTimeExtent	2469807.5 2475286.5
cdr:WRFRegionExtent	scale:hasTimeScale	scale:ConstantThroughTime

was created as a script for use with the statistical programming environment R (2011), requires several decades of historical weather data. For this modeling exercise the nearest station to provide the required data is at the Savannah/Hilton Head Airport. Data from 1949 to 2009 is used to generate the historical statistics. The model also accepts alternate values of daily average of temperature and precipitation, for each month. The produced data are post-processed to conform to the new averages. The time period modeled will be a 15-year span starting the year 2050.

The Savannah Airport weather model is defined to have two input data sets and eight output data sets. The output data sets have the “sw:SavannahAirportScale” as their data set scale, shown in Table 4-6. It defines the data as being at a point (the airport weather station, 32.147075 -81.210058) but being applicable for a 1 arc-degree space around the airport (geo:radius 1). The internal scale, being a point, is uniform. The time extent, defined with Julian dates, is from 1/1/2050 to 1/1/2065. For the input data (Table 4-7), it is looking for values at the location of the airport that are uniform through time and space. These are the monthly precipitation and temperature climate statistics.

4.3.4 Hydrology and Flatwoods Salamander Model

The salamander model was built using the NetLogo (Wilensky, 1999) platform. The NetLogo platform allows for the scripting of agent behavior and the interactions of the agent and the world around it. The Flatwoods Salamander model is set up to imitate salamander behavior.

Table 4-6. Scale triples for the output data sets defined as part of the Savannah Airport stochastic weather model. The radius for the airport is set to be 1 degree just for simplicity.

sw:SavannahAirportScale	scale:hasSpaceExtent	32.147075 -81.210058
sw:SavannahAirportScale	geo:radius	1

sw:SavannahAirportScale	scale:hasSpaceScale	scale:ConstantThroughSpace
sw:SavannahAirportScale	scale:hasTimeExtent	2469807.5 2475286.5
sw:SavannahAirportScale	scale:hasTimeScale	scale:HourlyScale

Table 4-7. Scale triples for the input data sets for the Savannah Airport stochastic weather model.

sw:SavannahAirportClimateDataScale	scale:hasSpaceExtent	32.147075 -81.210058
sw:SavannahAirportClimateDataScale	scale:hasSpaceScale	scale:ConstantThroughSpace
sw:SavannahAirportClimateDataScale	scale:hasTimeExtent	2469807.5 2475286.5
sw:SavannahAirportClimateDataScale	scale:hasTimeScale	scale:ConstantThroughTime

Not much is known about the salamanders, but it is estimated that they live roughly seven to ten years, staying near (less than 500m and probably less than 100m) and return to a natal pond for breeding. They live in crawfish or other animal burrows in the shallow subsurface and feed off of insects that cross their burrow. When the surface soil is saturated they emerge and migrate. During the fall or early winter if there is a strong cold-front they are signaled to return to their natal pond to breed and deposit eggs. If a sufficiently large storm or set of storms follows the cold-front signal to fill the ephemeral ponds for 12-15 weeks the eggs are able to hatch and grow and successfully transform into new salamanders. If the pond dries out before they complete the transformation process they desiccate and die. It is estimated that sufficient pond filling occurs with a seven-year return period, so there is not much margin of error for the salamanders. The ephemeral ponds vary in drainage area and shape and thus some years some will be full while others will have dried up. These ponds are very shallow. The subsurface consists of roughly two feet of sand with six feet of clay beneath the sand. In the pond areas the sand has a large fraction of organic matter as well.

The hydrology model created for the salamander model models ponds as individual bowls with a specified upland area (computed beforehand) that fills with precipitation and drains via a specified evapotranspiration loss rate.

The hydrology and Flatwoods Salamander model uses two data sets as input, a precipitation data set and a temperature data set. The data used by the model needs to be at

(a) (b)

Figure 4-21. Photos of ephemeral ponds at Fort Stewart. The change in vegetation in the background demarcates the transition from the pond to the neighboring "upland" area. The buttressing of the trees and the "knees" indicates the usual depth at which the pond fills. There are only slight differences in topography and soils between the pond land elevation and the neighboring terrain surface elevation. The terrain is hummocky and does not drain to a drainage network but rather drains to these ephemeral ponds. The (a) photo is of a pond, shown in Figure 4-8, where the salamanders were last sighted. The (b) photo is the pond labeled F9.5-01 in Figure 4-22. Photos taken by Aaron Byrd during a site visit on the 31st of January 2011. Subsurface investigations during the visit showed that in area (a) the soil was not fully saturated above the clay layer, while in (b) standing groundwater was encountered 4" below the surface. In both cases the impervious clay layer was approximately 2' below the surface. The buttressing at area (a) was approximately 4" while that in area (b) was approximately 2'.

daily scale. Further, while daily precipitation values are used, the temperature data is actually three daily values, the high, low, and mean daily temperature. The data will not only have to be converted to daily values but also need to be the three values over each day. The time series procedural knowledge base provides this facility. The input and output data set scale triples are shown in Table 4-8 and Table 4-9. The complete salamander model has five areas (of which this proof-of-concept uses just the first) and outputs data by individual ponds that have unique identifiers, such as F9.5-01. The pond shown in Figure 4-21 is part of the third area.

The agent-based model is also a stochastic model. For this proof-of-concept demonstration two salamander runs will be executed for each set of weather data. The first area has five salamander ponds; each is started with five salamanders at age zero. The maximum population for each pond is 100.

Table 4-8. The scale triples for the input data sets of the salamander model. The complete set of simulations has five salamander areas it models, hence this one is area one.

em:EM Area 1 Scale	scale:hasSpaceExtent	32.095650 -81.772045 32.070061 -81.739402
em:EM Area 1 Scale	scale:hasSpaceScale	em:Area 1 Internal Space Scale
em:Area 1 Internal Space Scale	scale:hasInternalScaleType	scale:UniformContinuous
em:EM Area 1 Scale	scale:hasTimeExtent	2469807.5 2475286.5
em:EM Area 1 Scale	scale:hasTimeScale	scale:DailyScale

Table 4-9. The scale triples for the output data set of the salamander model. F9.5-01 is the designation for a particular pond studied by wildlife biologists interested in the salamanders.

em:FWS F9.5-01 Scale	scale:hasSpaceExtent	F9.5-01 Extent
em:FWS F9.5-01 Scale	scale:hasSpaceScale	em:F9.5-01 Internal Space Scale
em:F9.5-01 Internal Space Scale	scale:hasInternalScaleType	scale:UniformContinuous
em:FWS F9.5-01 Scale	scale:hasTimeExtent	2469807.5 2475286.5
em:FWS F9.5-01 Scale	scale:hasTimeScale	scale:AnnualScale
F9.5-01 Extent	owl:SameAs	32.090011 -81.752433 32.091219 -81.753613

4.3.5 Project Specification

The project specification for this proof-of-concept has one decision data set, the salamander population for pond F9.5-01, and two alternative scenarios, a historic data scenario and an A1B climate scenario. The project is also set up to use reification triples (triples about triples, for example Table 4-10) in order to read in data sets and reference projects. These include the input WRF data sets, the Savannah Airport model data, and the salamander model data.

4.4 Results

The two scenarios run create five weather sets, each of which are run twice for the salamander model, results in a total of 20 simulation runs.

The results at the end of the 15-year period for all five salamander ponds in the salamander model are shown in Table 4-11 and are quite illustrative of the advantages to using this integrated modeling approach. The results, while only for preliminary data and of questionable statistical significance (this is a proof-of-concept model with only sufficient stochasticity to demonstrate that it is included in the concepts), show significant differences for three of the five ponds. Overall the A1B climate change data was actually helpful for the salamanders. After the fact the reason for this becomes clear – the climate data of Figure 4-19 forecasts more precipitation during the winter months which would tend to result in

Table 4-10. Initialization triple and reification statements used to denote a particular file to be read in as the WRF historic data.

cdr:FlatwoodsSalamander-ClimateChangeImpact	md:hasInitializationStatement	cdr:InitHistoricData
cdr:InitHistoricData	rdf:subject	wrf:WRFSimulation
cdr:InitHistoricData	rdf:predicate	cp:ReadProjectFile
cdr:InitHistoricData	rdf:object	C:\work\TestProjects\...

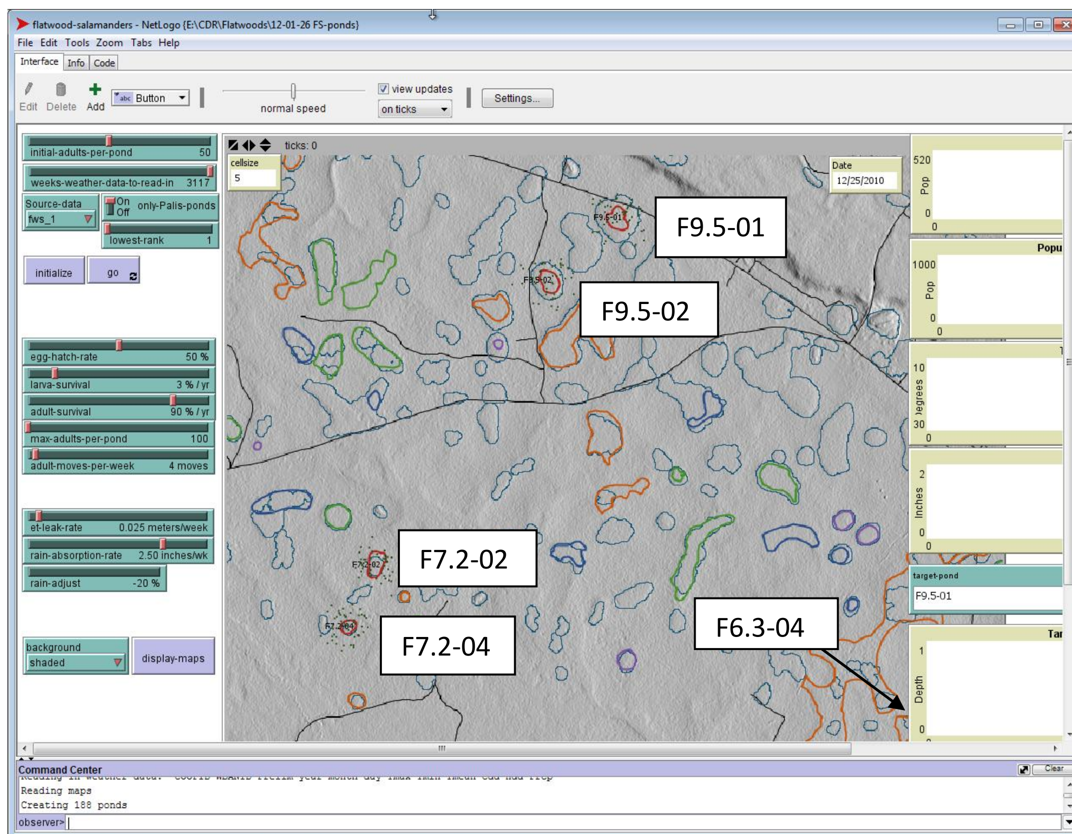


Figure 4-22. The Flatwoods Salamander model at initialization in the NetLogo environment. The model can either be run from the user interface or from the command line. The polygons with the dots are the salamander ponds – the dots are the salamander agents. For this image 50 salamanders are at each pond. For the actual runs only 5 salamanders started at each pond. This area is represented by the red square in Figure 4-8.

Table 4-11. Salamander population results of the two climate scenarios for 10 total runs each.

	A1B Climate Change		Historic Data	
	% chance of extirpation	Ave Pop Count of Viable Scenarios	% chance of extirpation	Ave Pop Count of Viable Scenarios
F6.3-04	0%	80.8	70%	6.3
F7.2-02	0%	43.6	100%	n/a
F7.2-04	0%	45.4	100%	n/a
F9.5-01	0%	88.6	0%	89.9
F9.5-02	0%	88.9	0%	69.8

ponds being more full and thus more likely to endure the full 15 weeks. The results also show that pond F9.5-01 is more robust than the others at providing sufficient hydrologic habitat, even in the face of adverse conditions.

More importantly, however, is that the results (and the output generated during the execution process) show that the model integration sequence worked as desired. The weather model created five hourly and five daily 15-year long data sets for each of the two scenarios. Ten input daily weather data sets and ten input experiment files were created for the salamander model; 20 summary output reports and 20 running population count reports were created by the salamander model. The climate model output monthly mean values in temperature and precipitation. The stochastic weather model used that data and produced sequences of rainfall, temperature, relative humidity, cloud cover, winds speed, etc. The hydrology and ecology model took that data and used it to drive pond depth over time as well as population dynamics. A quick summary of the total precipitation output for each weather set produced by the weather model (hourly data) matched that of the input weather precipitation data (daily data) indicating that the data was transformed correctly. A visual evaluation of the precipitation data shows that minimum, mean, and maximum values for each day were correctly calculated as well.

By using the semantic metamodel of computational models together with the data set semantic and procedural knowledge, the reasoning engine was able to successfully deduce the correct chain of models. The procedural knowledge also included checks to see if the output files exist so as to not duplicate the computational effort. Figure 4-23 shows part of the output log for created by the run and illustrates the interaction between finding the data sets and executing the models.

4.5 Discussion of Results

The results show that the reasoning engine, using the created functional ontologies, was able to a) deduce the sequence of models required to create the desired data set, b) deduce which specific simulations are spatially and temporally sufficient to provide the necessary data, and c) was able to read, interpret, and transform data from one model to another. In contrast to most workflow tools, the reasoning engine does not have a pre-set path to accomplish its goal but rather both deduces and executes the required work flow. The reasoning ability is able to deduce the work flow, while the procedural execution portions of the reasoning engine are able to execute the work flow.

Using the semantic metamodeling approach for computational models enabled two models, with different input and output data types and scales, input and output file formats, and model execution styles, to transform one set of data, the scenario input data, into the desired data used for decisions. The metamodel included procedural knowledge that operated on the concepts of the metamodel but that also translated the metamodel concepts to the semantic model concepts in order to successfully execute the individual computational models.

The key to deducing the sequence of models required, however, is the use of a metamodel of data sets. By providing immutable and partially mutable information about the data sets, the data set functional ontology, by using the auxiliary functional ontologies of scale, time, and space, can deduce which data sets can be derived from others. This deductive ability coupled with the computational model execution abilities transforms the semantic and procedural reasoning engine into a powerful deductive workflow engine that is able to capture and utilize a wide variety of knowledge creation and deduction mechanisms.

The two models used for this integration exercise, one based on R and the other on Java, illustrate how this approach can integrate very different types of computational models. The goal of this effort was not to create a model coupling methodology or standard, but rather to demonstrate the concept that a functional ontology reasoning engine, with functional ontologies about data and models, can deduce the need to couple the models, choose models that are able to compute the desired data at the appropriate scale and location, execute them, and transform data as needed. The example project, while done at a proof-of-concept level, illustrates a very real use case with real-life complexities of different data types and internal scales, non-spatially congruent model boundaries, and different file formats. The example project demonstrates that semantic and procedural modeling using semantic metamodels is a complimentary technology for computational models.

4.6 Conclusions

Semantic and procedural knowledge modeling is a very complimentary tool to the computational models we use as hydrologists. The semantic metamodeling approach allowed for disparate models to be brought into the semantic reasoning process in order to deduce one type of knowledge from another. The reasoning engine was able to first deduce that a model integration exercise was required. Next, it was able to select an appropriate set of models based on the description of the characteristics of the data that they each require and generate. Finally, it was able to execute the models and transform the data as required.

The semantic and procedural reasoning engine and knowledge base is a powerful tool for capturing and utilizing the knowledge we have as hydrologists to automate and facilitate many of the tasks we do.

```
Read 20 ontologies
Compiling code to C:\Users\Administrator\AppData\Local\Temp\pwjljsoo.nbj\fos_d63057e5-c688-4cc5-a25f-2ff53bf8804e.dll
Successfully compiled the code.
Found 4 full initialization statements for project cdr:FlatwoodsSalamanderClimateChangeImpact
Deduced that the data set cdr:FtStewartFWSPopulationCount can be created for scenario cdr:A1BClimateForcing from reference model C:\work\TestProjects\ModelIntegration\Flatwoods\testrun\integrationrun.xml:track-pop-for-pond1-test
Deduced that the data set em:FtStewart Temperature - Area 1 can be created for scenario cdr:A1BClimateForcing from reference model sw:SavannahAirportModel
Deduced that an existing scenario data set fulfills the requirements for input data set sw:SavannahAirportClimateTempData for scenario cdr:A1BClimateForcing
Deduced that an existing scenario data set fulfills the requirements for input data set sw:SavannahAirportClimatePrecipData for scenario cdr:A1BClimateForcing
All 2 input data sets exist to execute reference model sw:SavannahAirportModel for scenario cdr:A1BClimateForcing
Reading output data from sw:SavannahAirportModel, scenario cdr:A1BClimateForcing
Read 8 data sets (time series) from each of 6 output files
Successfully computed input data set em:FtStewart Temperature - Area 1 for scenario cdr:A1BClimateForcing
Deduced that an existing scenario data set fulfills the requirements for input data set em:FtStewart Precipitation - Area 1 for scenario cdr:A1BClimateForcing
All 2 input data sets exist to execute reference model C:\work\TestProjects\ModelIntegration\Flatwoods\testrun\integrationrun.xml:track-pop-for-pond1-test for scenario cdr:A1BClimateForcing
Converting time scales for four data sets to create weather data...
All output data sets already computed for FWS experiment, reading results
Reading C:\work\TestProjects\ModelIntegration\Flatwoods\testrun\final-counts\report-cdr_A1BClimateForcing_1__track-pop-for-pond1-test-1.txt
Reading C:\work\TestProjects\ModelIntegration\Flatwoods\testrun\final-counts\report-cdr_A1BClimateForcing_1__track-pop-for-pond1-test-2.txt
…
```

Figure 4-23. Part of the log output by the procedural knowledge of the several ontologies created for this work.

References

De Virgilio, R., 2010. Meta-Modeling of Semantic Web Services, Services Computing (SCC), 2010 IEEE International Conference on, pp. 162-169.

GeoRSS, 2012. GeoRSS Simple Model. In: Consortium, O.G. (Ed.). http://www.georss.org/simple.

Jarke, M., Jeusfeld, M.A., Nissen, H.W., Quix, C., 2009. Heterogeneity in model management - a meta modeling approach. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (Eds.), Conceptual Modeling: Foundations and Applications, Essays in Honor of John Mylopoulos. Springer-Verlag: Berlin, pp. 237-253.

Jeusfeld, M.A., 2009. Metamodeling with Datalog and Classes: ConceptBase at the Age of 21. Proceedings Intl. Conf. on Object Databases (ICOODB-2009): Zurich, Switzerland.

Michalakes, J., Dudhia, J., Gill, D., Henderson, T., Klemp, J., Skamarock, W., W. Wang, 2004. The Weather Reseach and Forecast Model: Software Architecture and Performance. In: Mozdzynski, G. (Ed.), Proceedings of the 11th ECMWF Workshop on the Use of High Performance Computing In Meteorology: Reading, U.K.

Nakićenović, N., Swart, R., 2000. Special Report on Emissions Scenarios: A special report of Working Group III of the Intergovernmental Panel on Climate Change. Cambridge University Press, Cambridge.

Oracle, 2011. Java(TM) Standard Runtime Environment (1.6.0_23) http://www.java.com

Payne, T., Lassila, O., 2004. Semantic Web Services. IEEE Intelligent Systems 19(4), 14-15.

R, 2011. R: A language and environment for statistical computing. R Foundation for Statistical Computing: Vienna, Austria.

UML, 2010. OMG Unified Modeling Language(TM) (OMG UML) Infrastructure. Object Management Group. http://www.uml.org. (accessed December 15, 2011).

Westervelt, J., Sperry, J., Burton, J., Palis, J., 2012. Modeling response of flatwoods salamander populations to historic and predicted climate variables. Ecological Modeling, in review.

Wilensky, U., 1999. NetLogo. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

CHAPTER 5

FUNCTIONAL ONTOLOGY CLASS METHODS AND USAGE

This chapter serves as the reference manual for the reasoning engine. It presents formal definitions for functional ontology classes and methods and the specifications for structuring code in the functional ontology framework. It discusses the details of how code is integrated into the reasoning engine, how the data code is wrapped, how the reasoning engine works, and the classes and class members and methods of the reasoning engine API that can be used in an external program or in the data code.

The goal of a functional ontology knowledge base is to encode both semantic knowledge (i.e. concepts and relationships between them) as well as procedural knowledge directly associated with the concepts. The functional ontology reasoning engine is the core set of code that is able to execute semantic queries and procedural calls, together. The reasoning engine is a library that can be used in any C# program (or other Microsoft .NET framework language) and is separate from the graphical user interface (GUI) demonstrated in Chapter two. The reasoning engine can thus be added to any program and not just used with the GUI. This document outlines the general operation of the reasoning engine, as well as the classes and methods of the reasoning engine library. The classes and methods are usable in both an external code that creates an instance of a functional ontology reasoning engine as well as the code in any functional ontology knowledge base.

5.1 Functional Ontology Coding Framework

The functional ontology API is built on the Microsoft® .Net Framework. The .Net framework is built on a unique technology, called Common Language Infrastructure (ISO,

2006), where code is first compiled to a low-level language (called Common Intermediate Language Infrastructure, CIL) and then, as it needs to execute the code, uses a Virtual Execution Engine (VEC) technique to turn the CIL code into executable instructions. This method allows the code base at compile time to be extended later at run time. Apart from facilitating user-readable code in the ontologies, this allows for the knowledge base in use by the reasoning engine to grow during the reasoning process, including the potential of self-coding algorithms.

The functional ontology API can read in an RDF/XML file. The file, being XML, allows for CDATA nodes. The functional ontology language specifies a set of predicates that have as objects CDATA structures containing code. This code, which I will term “code data,” once read in to the API, has to be wrapped in code to create a complete source code structure that can be compiled into a library. The API has defined source code headers to wrap the code data. The code data can access some forms of other code data, such as common code, by using the headers while some forms of code data, such as predicate methods, are best accessed by invoking functionality of the API. The following sections detail how the code is wrapped and how to access it from other code data.

5.1.1 Key Concepts

The reasoning engine links the code data (in the ontologies) to the reasoning engine by wrapping the code snippets to create a set of fully defined c# classes (objects), compiling them into a dynamic link library (DLL), creating an instance of them (live version,) and finally associating the classes in the DLL to the concepts in the reasoning engine . The reasoning engine itself is a DLL has several classes that the main reasoning engine class (FuncOnt) uses,

such as BaseOntNode and NamedNodeSet. The following sections go into more detail about the individual classes, members and methods, and how the reasoning engine works.

5.1.2 Integrating Code Data into the API

Once the code data is read in to the API and placed in the code wrapping classes the complete set of source code is compiled into a dynamic link library (DLL) with the c# compiler built into the .NET Framework. A new application domain is created and the DLL is attached to the application domain. An application domain is the conceptual housing for a running program in the operating system. The goal of the separate application domain is to allow for the graceful failure of the code data DLL without crashing the entire reasoning engine. There are two peculiarities, however, about this method. If the reasoning engine application domain directly calls the methods of the data code application domain, the data code DLL is instantiated as part of the reasoning engine application domain, at which point the failure of the data code DLL will crash the entire reasoning engine. To overcome this, a set of interfaces are created that include methods to call a named method of the class. In Figure 5-3, the complete code listing for a predicate function, shows a class that inherits from two interfaces, IRemoteInterface and IPredicateInterface, and has two methods, Invoke and EvalPredicate. The IRemoteInterface defines the Invoke method header while the IPredicateInterface defines the EvalPredicate method header. The API calls the Invoke method with the EvalPredicate method name, and an object list of the required parameters, when appropriate. The API uses a remote creation factory to create classes in the data code application domain, which are only referenced in the reasoning engine API via the IRemoteInterface. This brings us to the second peculiarity of this approach to having two application domains, accessing memory from one in the other.

The only method that will allow for code in one application domain to access memory in the other is to create classes that inherit from the class "MarshalByRefObject," which is also shown in Figure 5-3. C# restricts class inheritance to single inheritance from classes but multiple inheritance from interfaces. Thus all classes in the API need to inherit from MarshalByRefObject. There are two primary classes that are used by programs using the API, the FuncOnt class, which is the functional ontology reasoning engine, and the NamedNodeSet class, which contains the results lists. The NamedNodeSet class is a container with a dictionary for the linked list of nodes for the sets created by queries. It would be effective to have it inherit from the dictionary class, but since it must inherit from the MarshalByRefObject it therefore contains a dictionary object, called set. The procedural code can create temporary NamedNodeSet objects that the reasoning engine can use, and the procedural code can examine, by calling a create function on the FuncOnt class.

5.1.3 Naming Convention

Inside the reasoning engine class, the concept names area stored in "string" variables. The string variables holding the concept names are able to hold any character data. The linking of the code data to the concept name for the code involves wrapping the code in a complete class heading and invocation methods. The name of the concept is used to create the internal name of the class holding the code. The user needs to know the naming convention because the source code in the knowledge base is able to access other code in the knowledge base. This allows for a broad range of programming styles. Turning the concept names into code, however, involves some restrictions on possible characters. For instance, the colon, any spaces, or any punctuation in the concept string are not permissible as class or method names in the code file. In the creation of class or method names from the

concept strings all invalid characters are replaced with an underscore. For example, “ex:My Cool Concept” would become “ex_My_Cool_Concept.”

5.1.4 Section Overview

Section 5.2 describes the Functional Ontology semantic keywords as well as how they relate to code parts. Section 5.3 details how the reasoning engine and a user interface are able to interact. Section 5.4 covers the methods and members of the two primary classes in the reasoning engine API, the FuncOnt class (the reasoning engine class) and the NamedNodeSet class (the class that holds the answer sets from the reasoning engine).

5.2 Functional Ontology Code Data and Language Reference

The functional ontology language is described in the Chapter two and includes details about including source code in a functional ontology knowledge base. The code in the knowledge base is the functional code for the concepts but is insufficient for direct use. The knowledge base code (code data) must first be wrapped in a class header and invocation methods to create a complete code base that is then compiled and instantiated by the reasoning engine. The functional ontology reasoning engine class is, in general, passed to the methods encapsulating the knowledge base code (code data) so that the code in the knowledge base has complete access to any and all methods of the reasoning engine, such as the ability to further query the reasoning engine.

The following sections describe in more detail how the code is wrapped and referenced for the various code data types.

5.2.1 fo:PrimaryCode

The fo:PrimaryCode is the predicate used to identify the data code that functions as the fallback procedure for the reasoning engine. When the reasoning engine is not able to find a match for the query, or when expressly directed to, it will execute the primary code. For example, the RDF/XML file could have the following primary code, Figure 5-1.

Figure 5-1 shows some data code for a predicate “poly:hasArea.” The code, when read in, is added to the triple store but also added to a class (internal to the API) that creates the wrapper code for the method. The API has a set of classes that are used to store the data code and in turn generate the wrapped code that will shortly be compiled. The header for the actual method called for the predicate code is shown in Figure 5-2. The subject,

```
double a = 0, b = 0;
double[] x;
double[] y;
int count;
bool hasPoints=poly_GetPoints.getPoints(theSubject, out x, out y, out count,
                                        theOntology);

if (!hasPoints)
  return 0; // couldn't complete the computation of the area

for (int i = 0; i < count; i++)
{
  a = a + (x[i] * y[i + 1]);
  b = b + (y[i] * x[i + 1]);
}
double twicearea = a - b;
if (twicearea < 0.0)
  twicearea = twicearea * -1.0;
double area = twicearea / 2.0;

theOntology.AddTriple(theSubject, thePredicate,string.Format("{0:0.00}", area));

return theOntology.FindMatchingSet(theSubject, thePredicate, theObject, results);
```

Figure 5-1. Data code for the poly:hasArea predicate function. This code computes the area of a polygon.

```
public int EvalPredicate(string theSubject, string thePredicate, string theObject,
                         FuncOnt theOntology, NamedNodeSet results)
```

Figure 5-2. Method header for the predicate procedure. The query is of the form (Subject, Predicate, Object, results) so those are passed to the method. FuncOnt is the Functional Ontology class reference.

predicate, object, and results class are passed in as parameters, along with a reference to the ontology class making the call. The return value is the return value of the query, meaning the count of concepts matching the query. Zero is returned for an error or if no value is created or found. The predicate method is able to call any method on the functional ontology class.

The EvalPredicate method shown in Figure 5-2 is a member of a class created to hold all the code related to the predicate. Separate classes are created for each predicate. The class name in the code will be the underscore version of the predicate's string. In this example that would be "poly_hasArea." The full class code for the wrapped function is shown in Figure 5-3.

The predicate code data can refer to the five parameters. The parameters "theSubject," "thePredicate," and "theObject" refer to the triple query. The parameter "theOntology" is a reference to the reasoning engine class. The parameter "results" is the NamedNodeSet. The predicate code can either create the linked of concept nodes and make the set or it can, as shown in Figure 5-1 and Figure 5-3, add triples and rerun the query. To call the predicate function, execute a query with the predicate you want to call. To force a query to call the predicate function prepend a "+" to the predicate concept ("A +P ?B"). Alternatively, if you do not want a query to call a predicate function prepend a "-" to the predicate concept ("A –P ?B").

5.2.2 fo:SecondaryCode

Secondary code is the term used in the functional ontology for members of the predicate or user function that are not the primary method. For the predicate code in Figure 5-3 the primary method is the "EvalPredicate" method. The secondary code should include all the method headers and may contain more than just one method.

```
public class poly_hasArea : MarshalByRefObject, IRemoteInterface, IPredicate
{

  public object Invoke(string MethodName,object[] Parameters)
  {
    return this.GetType().InvokeMember(MethodName, BindingFlags.InvokeMethod,
                                       null, this, Parameters);
  }
  public int EvalPredicate(string theSubject, string thePredicate,
                           string theObject, FuncOnt theOntology,
                           NamedNodeSet results)
  {

    double a = 0, b = 0;
    double[] x;
    double[] y;
    int count;
    bool hasPoints=poly_GetPoints.getPoints(theSubject, out x, out y, out count,
                                            theOntology);

    if (!hasPoints)
      return 0; // couldn't complete the computation of the area

    for (int i = 0; i < count; i++)
    {
      a = a + (x[i] * y[i + 1]);
      b = b + (y[i] * x[i + 1]);
    }
    double twicearea = a - b;
    if (twicearea < 0.0)
      twicearea = twicearea * -1.0;
    double area = twicearea / 2.0;

    theOntology.AddTriple(theSubject, thePredicate,string.Format("{0:0.00}",
                          area));

    return theOntology.FindMatchingSet(theSubject, thePredicate, theObject,
                                       results);
  }
}
```

Figure 5-3. Complete code wrapping the “poly:hasArea” predicate method.

```
public bool DoesTerrainGroupHaveActualData(string TerrainGroup,string ActualDataType,
  FuncOnt theOntology)
{
NamedNodeSet tmpResults = theOntology.MakeTempNamedNodeSet();
theOntology.FindMatchingSet(TerrainGroup, "td:hasActualDataAssociation",
                            "?allData", tmpResults);
if (!tmpResults.ExistsSet("allData"))
  return false;
int retval = theOntology.FindMatchingSet(tmpResults.set["allData"],"rdf:type",
  ActualDataType,tmpResults);
if (retval>0)
  return true;
else
  return false;
}
```

Figure 5-4. RDF/XML data for secondary code. The primary predicate is “td:hasComputableDependents.”

5.2.3 fo:CommonClass

The common class code is meant to hold code that can be accessed by many other classes and should be reusable. Common class definitions are created as triples. The subject of the triple, after illegal characters are changed to underscores, will be used to create the class name. Figure 5-6 shows an example of a common code definition in RDF/XML, while Figure 5-7 shows the code as it is wrapped for the DLL. Notice that in the RDF/XML code (Figure 5-6) the class header and brackets are not explicitly defined. The rest of the class, including any members or methods, are defined in the RDF/XML code as it would appear in a source code file. This method is called in both Figure 5-1 and Figure 5-3. In this example the method is declared static, meaning that it can be accessed without creating an instance of a class. The common class could also be one that is used as instances in the other code.

5.2.4 fo:UserCode

The user code is meant to be a procedure that a user would want to call directly. For example, it could be called to accomplish a specific, over-arching task such as delineating a watershed or calculating the area of all polygons.

5.2.5 fo:UsingFile

The using file predicate is meant to add a library to the project. Both the standard libraries and user-made libraries can be added. It is usually used in conjunction with the using namespace predicate. While a file can be specified for each code set, the file will only be included once in the project. All file inclusions apply globally.

5.2.6 fo:UsingNamespace

The using namespace predicate adds the namespace to the project. The object of the predicate is the namespace without the “using” keyword or a semicolon. Like the using file predicate, each code base can have any number of using namespace predicates but they will only be included once. All namespace inclusions apply globally.

5.3 Reasoning Engine API and User Interface Integration

The reasoning engine library (i.e. application programming interface, API) contains the core functionality for reasoning over semantic and procedural triples as well as compiling and executing the code in the procedural triples.

Figure 5-10 shows a simple integration between the reasoning engine API and a user interface. At the heart of it the reasoning engine is simply a class that is created by the user interface. The concepts, relationships, and code are then fed to the class and the knowledge base source code is compiled (either automatically or when the user interface code instructs it to do so.) The reasoning engine then waits for the user interface to invoke a method, such as performing a semantic query.

There are five primary interaction points (possible operations) between the reasoning engine class and the external program. In the diagram below these are the operations available. These are, generally speaking, reading data (from file or memory), writing data, performing a semantic query, executing a user function, and compiling (or recompiling) the knowledge base code,

The first operation or interaction point is the query method, FindMatchingSet(). The query routine will both execute a logic-based search as well as execute predicate functions, if necessary. The second interaction point is in the execution of user functions, RunUserFunction(). There are also methods to retrieve the list of concepts that have

attached user functions. The next three interaction points are for input/output. There are two methods of reading the triple store, one is via a RDF/XML file and the other is via an in-memory container. The in-memory container is designed so that it can be viewed by a C# program that is interacting with the reasoning engine API. Finally there is a method to recompile all the code and a boolean flag that controls if the code is recompiled immediately after reading in a file. This flag, RecompileOnDemand, should be true if, for example, a

```
public class td_hasComputableDependents : MarshalByRefObject,
                                          IRemoteInterface, IPredicate
{

  public object Invoke(string MethodName, object[] Parameters)
  {
    return this.GetType().InvokeMember(MethodName, BindingFlags.InvokeMethod,
                                   null, this, Parameters);
  }
  public int EvalPredicate(string theSubject, string thePredicate,
                           string theObject, FuncOnt theOntology,
                           NamedNodeSet results)
  {
      …
  }
  public bool DoesTerrainGroupHaveActualData(string TerrainGroup,
       string ActualDataType, FuncOnt theOntology)
  {
    NamedNodeSet tmpResults = theOntology.MakeTempNamedNodeSet();
    theOntology.FindMatchingSet(TerrainGroup, "td:hasActualDataAssociation",
                            "?allData", tmpResults);
    if (!tmpResults.ExistsSet("allData"))
      return false;
    int retval = theOntology.FindMatchingSet(tmpResults.set["allData"],
                                 "rdf:type", ActualDataType, tmpResults);
    if (retval > 0)
      return true;
    else
      return false;
  }
}
```

Figure 5-5. Wrapped code for the secondary code “DoesTerrainGroupHaveActualData” of the primary predicate “td:hasComputableDependents.” The RDF/XML code data is shown in Figure 5-4.

wrapping program is reading and setting up a few ontology stores before sending the data to the reasoning engine API.

```
public static bool getPoints(string thePoly, out double[] x, out double[] y,
      out int count, FuncOnt theOntology)
{
  NamedNodeSet tmpResults = theOntology.MakeTempNamedNodeSet();
  bool retval = (theOntology.FindMatchingSet(thePoly,"poly:hasPointString",
       "?thepoints",tmpResults)>0);
  if (retval)
  {
    string points = tmpResults.First("thepoints");
    char[] pointseparator = { '(' };
    string[] pointlist = points.Split(pointseparator,
        StringSplitOptions.RemoveEmptyEntries);
    char[] numseparator = { ' ', ',', ')' };
    count = pointlist.Length;
    x = new double[count + 1];
    y = new double[count + 1];
    for (int i = 0; i < count; i++)
    {
      string[] data = pointlist[i].Split(numseparator,

StringSplitOptions.RemoveEmptyEntries);
      x[i] = Convert.ToDouble(data[0]);
      y[i] = Convert.ToDouble(data[1]);
    }
    x[count] = x[0];
    y[count] = y[0];
  }
  else
  {
    x = new double[1];
    y = new double[1];
    x[0] = 0.0;
    y[0] = 0.0;
    count = 0;
  }
  return retval;
}
```

Figure 5-6. Common code example RDF/XML example. Note that this code contains everything but the class definition statement and brackets.

```
public class poly_GetPoints
{
  public static bool getPoints(string thePoly, out double[] x, out double[] y,
                               out int count, FuncOnt theOntology)
  {
    NamedNodeSet tmpResults = theOntology.MakeTempNamedNodeSet();
    bool retval = (theOntology.FindMatchingSet(thePoly,"poly:hasPointString",
                                             "?thepoints",tmpResults)>0);
    if (retval)
    {
      string points = tmpResults.First("thepoints");
      char[] pointseparator = { '(' };
      string[] pointlist = points.Split(pointseparator,
                                  StringSplitOptions.RemoveEmptyEntries);
      char[] numseparator = { ' ', ',', ')' };
      count = pointlist.Length;
      x = new double[count + 1];
      y = new double[count + 1];
      for (int i = 0; i < count; i++)
      {
        string[] data = pointlist[i].Split(numseparator,
                                  StringSplitOptions.RemoveEmptyEntries);
        x[i] = Convert.ToDouble(data[0]);
        y[i] = Convert.ToDouble(data[1]);
      }
      x[count] = x[0];
      y[count] = y[0];
    }
    else
    {
      x = new double[1];
      y = new double[1];
      x[0] = 0.0;
      y[0] = 0.0;
      count = 0;
    }
    return retval;
  }

}
```

Figure 5-7. Wrapped common code from Figure 5-6. The class definition statement was created from the name of the defined predicate.

```
  string newgroup = theOntology.context.ValueOf("td:CurrentTerrainGroup");
  string outlet=ccwpf_GetUserString.Get("Open Outlet File","Please enter the file name
 for the outlet shapefile.");
  if (cc_DoesFileExist.DoesFileExist(outlet)) {
    theOntology.AddTriple(outlet,"rdf:type","td:Outlet");
    theOntology.AddTriple(newgroup,"td:hasActualDataAssociation",outlet);
  theOntology.context.AddValue("td:CurrentOutlet",outlet);
  } else {
    theOntology.log.AddLog("Error: unable to find file. Please re-run function");
  }
```

Figure 5-8. RDF/XML user code for an “Add Outlet” procedure.

```
public class td_AddOutlet : MarshalByRefObject,IRemoteInterface,IUserFunction
{
  public object Invoke(string MethodName,object[] Parameters)
  {
    return this.GetType().InvokeMember(MethodName,
       BindingFlags.InvokeMethod,null,this,Parameters);
  }
  public void UserMain(FuncOnt theOntology, NamedNodeSet results)
  {

  string newgroup = theOntology.context.ValueOf("td:CurrentTerrainGroup");
  string outlet=ccwpf_GetUserString.Get("Open Outlet File","Please enter the
 file name for the outlet shapefile.");
  if (cc_DoesFileExist.DoesFileExist(outlet)) {
    theOntology.AddTriple(outlet,"rdf:type","td:Outlet");
    theOntology.AddTriple(newgroup,"td:hasActualDataAssociation",outlet);
  theOntology.context.AddValue("td:CurrentOutlet",outlet);
  } else {
    theOntology.log.AddLog("Error: unable to find file. Please re-run
function");
  }

  }
}
```

Figure 5-9. The wrapped user code shown in Figure 5-8.

5.4 API Classes

There are several classes in the API but only two that are instantiated outside of the API itself: the reasoning engine class, FuncOnt, and the query results class, NamedNodeSet. FuncOnt is the primary class of the reasoning engine API. The members and methods hold

and operate on a network of concept nodes. All information about a concept is stored in its concept node, which is an instance of the class BaseOntNode. Concept nodes that are the results of queries are referenced in linked lists in a NamedNodeSet. The NamedNodeSet class holds dictionaries of set names and linked lists of the values, and also includes set operators for the lists. BaseOntNodes stores different lists of relationship types.

5.4.1 BaseOntNode Member and Method Description

BaseOntNode class instances hold the identifying string for the concept as well as lists for the possible logic connections. Currently these logic connections include superclass/subclass, superproperty/subproperty, inverse, equivalence, domain, and range logic types. It also holds a list of all the triples it is defined to be a part of, as well as an alternate form of the triples called a base node pair. A base node pair consists of the other two concepts to form a triple as well as a position indicator (subject, predicate, object) for the root concept. The reasoning engine loops over these lists to search for potential matching triples as well as the possible logic connections. These lists are created automatically by the AddTriple procedure of the functional ontology (FuncOnt) class. The only method used outside of the functional ontology class is the ToString() method, which returns a string version of the concept.

5.4.2 NamedNodeSet Member and Method Description

Named node sets are the answer sets for the queries. Queries of the form <?A B C>, <A ?B C>, or <A B ?C> are in essence asking for a set of concepts that makes the statements true. The name of the set is the string after the question mark.

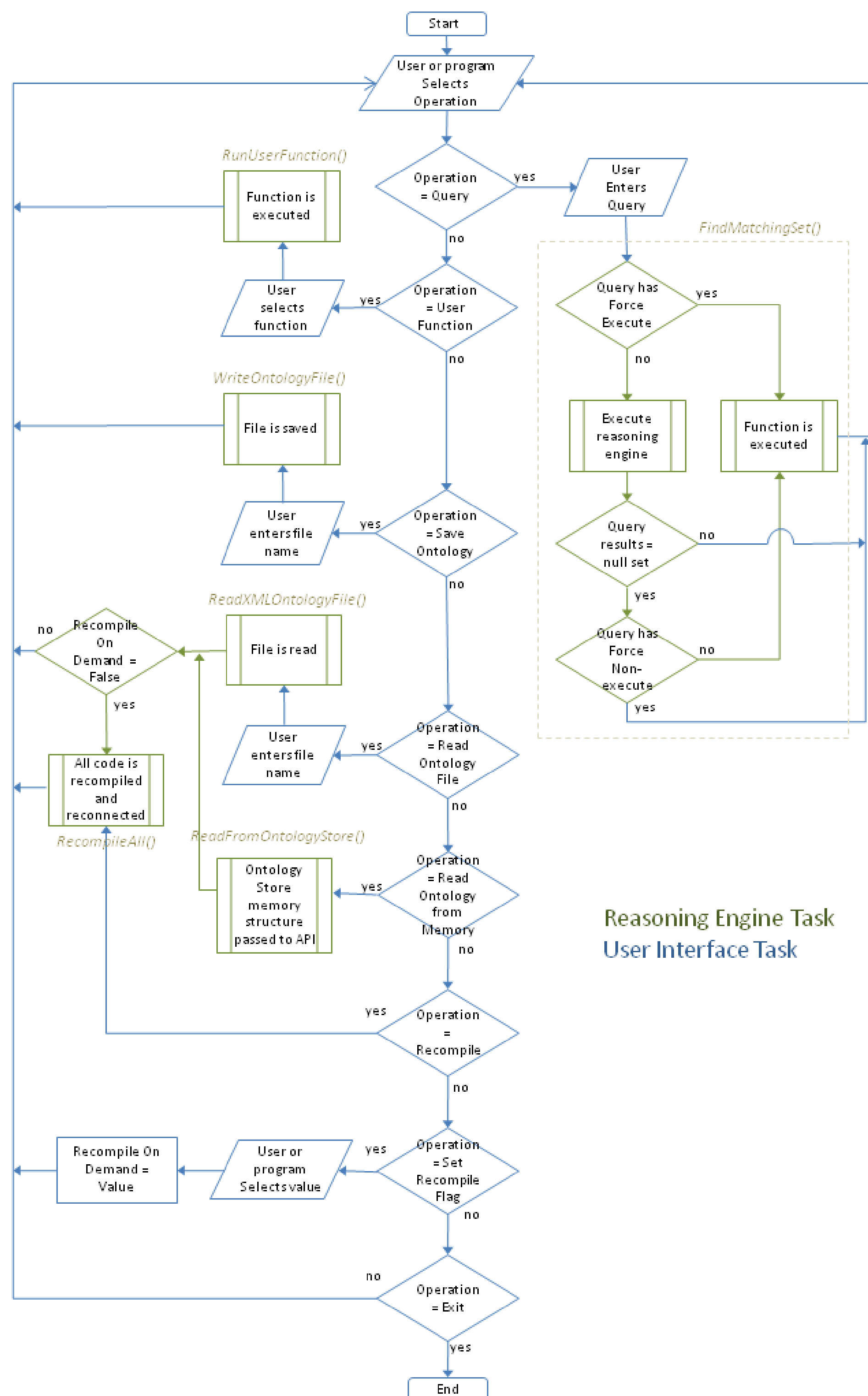


Figure 5-10. Possible interaction mechanisms between the reasoning engine class and a user interface.

5.4.2.1 NamedNodeSet::set

The only member of a named node set is called “set.” Since C# only allows single inheritance and instances of NamedNodeSets need to inherit from the class “MarshalByRefObject” in order to facilitate cross-application domain memory access, the “set” member is the main set storage mechanism. The “set” member is a dictionary that relates a string keyword (in this case the set name from the query) to a linked list of BaseOntNodes. As a “Dictionary,” the set member has all the methods of the “Dictionary” class, such as “count,” “Remove,” “Add,” “ContainsKey,” etc.

5.4.2.2 bool NamedNodeSet::SetContains(string setname, string nodename)

The SetContains function looks through the nodes of the linked list associated with the setname and determines whether or not nodename belongs to the list.

5.4.2.3 int NamedNodeSet::Union(string list1, string list2, string result)

The Union operator takes two lists and creates a third list that consists of all the unique concept members of both sets. The resulting list name is indicated by result. A set Union operation is equivalent to a logical “or” operation. Returns the count of the result set.

5.4.2.4 int NamedNodeSet::Intersection(string list1, string list2, string result)

The intersection operator takes two sets, list1 and list2, and determines which concepts are in both of the lists. The set of concepts that belong to both lists is created and given the name indicated by result. A set intersection operation is equivalent to a logical “and” operation. Returns the count of the result set.

5.4.2.5 int NamedNodeSet::Difference(string BigSet, string SubtractSet, string result)

A set difference operation is similar to subtracting one from the other. All elements of SubtractSet are removed from BigSet, if they are a part of BigSet. The resulting set of elements is put in the set named by results; BigSet is not changed during the operation. Returns the count of the results set.

5.4.2.6 int NamedNodeSet::Size(string name)

Returns the number of elements in the set name.

5.4.2.7 void NamedNodeSet::Merge(NamedNodeSet other)

The merge method takes another NamedNodeSet and adds the sets to the current sets. If any sets have the same key name then the result is the union of the two sets.

5.4.2.8 string NamedNodeSet::Remove(string name)

Removes the set indicated by name from the list of sets.

5.4.2.9 string NamedNodeSet::First(string name)

Returns a string of the first element of the list indicated by name.

5.4.2.10 bool NamedNodeSet::ExistsSet(string name)

Checks to see if the set indicated by name is a valid key in the set member dictionary.

5.4.2.11 bool NamedNodeSet::ExistsNonEmptySet(string name)

Checks to see if the set indicated by name is a valid key in the set member dictionary and the size of the set is greater than zero.

5.4.2.12 void NamedNodeSet::AddToSet(string setname, string concept, FuncOnt theOntology)

Adds the BaseOntNode denoted by the parameter concept to the set indicated by setname. If the set does not exist then it creates the new set.

5.4.2.13 void NamedNodeSet::AddListToSet(LinkedList<BaseOntNode> list, string name)

Adds the list to the set under the name indicated by name. If there exists a set by the same name then the existing set is removed.

5.4.3 FuncOnt Member and Method Description

The FuncOnt class is the primary reasoning engine class. The members of the FuncOnt class are used to hold the information from the knowledge base. All of the concepts and triples are stored in one list but they source ontology for each triple (or sets of triples) can be different and is kept track of. Writing of the triples can be done over all the triples or a single source. There are a few members of the FuncOnt class: identList is a hybrid dictionary of the concept nodes (BaseOntNodes). This is the master list of all concept nodes. theTripleList is the master list of the triples. UserFunctions is the list of all specified user functions. PredFuncList is the list of all predicate functions. ScriptEngine is the class the handles code wrapping and compiling the code. NamespaceList is the list of all the namespaces referenced by the ontologies. NamedOntologyNodes is a set of triple collections. It is set up to be able to be linked to C#/XAML GUI objects for display. RecompileOnDemand is a key flag that controls whether or not the ScriptEngine compilation process is invoked immediately after reading in a file or memory store. If more than one file or memory store is to be read it should be set to true (to not invoke the ScriptEngine

compilation process). OntologyNameSet is the list of all ontology names. The name is changed when files are read in. The default name is “local.rdf.” CurrentOntologyName is a property that can be both set and retrieved. CurrentTempDirectory is also a property that can be set or retrieved. CurrentTripleCollection is for external C#/XAML display purposes. log is the collection of statements added by procedural code or the ScriptEngine.

5.4.3.1 FuncOnt Constructor

There are two constructors: FuncOnt() : this("local.rdf") and FuncOnt(string InitialOntologyName): base(). The version that takes doesn’t take an initial ontology name calls the other to set up the default name, “local.rdf.” The version that takes the initial ontology name also sets up the temporary directory for the script engine and some of the BaseOntNodes for the logic relationships.

5.4.3.2 NamedNodeSet FuncOnt::MakeTempNamedNodeSet()

The MakeTempNamedNodeSet method is used to create a NamedNodeSet in the data code application domain. Temporary NamedNodeSets are frequently used in the data code in conjunction with the method calls to the reasoning engine.

5.4.3.3 LinkedList<BaseOntNode> FuncOnt::MakeBaseOntNodeList()

The MakeBaseOntNodeList method is used to create a linked list class for BaseOntNodes in the data code application domain.

5.4.3.4 string FuncOnt::EncodeLiteral(string value)

The EncodeLiteral method is used to create a reference concept from a literal value. The literal value could be any character string. This method allows for identical literal values

to represent unique concepts in an ontology. The method of encoding the literal string is to format it a part of the following string "[GUID:literal]." The GUID ensures uniqueness of the concepts.

5.4.3.5 string FuncOnt::DecodeLiteral(string encodedvalue)

The DecodeLiteral method reverses the encoding of the EncodeLiteral method. If the encoded value is of the form "[string:literal]" then it will return the literal portion. If it is not then it will return the original value.

5.4.3.6 bool FuncOnt::AlreadyInList(string ID)

AlreadyInList checks the identList to see if it contains the key ID.

5.4.3.7 AddTriple Methods

There are two AddTriple methods. The primary method that adds a triple to the ontology base is "void AddTriple(string, string, string)." The other version, "void AddTriple(string SubjectConcept, string PredicateConcept, string ObjectConcept, NamedNodeSet results)," checks to see if any of the triple concepts are prepended with a "+" character. If so, the concept is treated as the name of a set in the results set. Each member of the set is then substituted into the triple and the other AddTriple method is called.

5.4.3.8 FindMatchingSet Methods

The FindMatchingSet method is the primary query function. There are several versions of the FindMatchingSet function. The actual method that accomplishes the query is FindStandardMatchingSet. The return value is the count of true triples deduced.

To identify the key search field prepend the field with a question mark (?). To force the predicate function to be called prepend the predicate with a plus sign (+). To ensure that the predicate function is not called prepend the predicate with a minus sign (-).

5.4.3.8.1 int FuncOnt::FindStandardMatchingSet(string Sub, string Pred, string Obj, NamedNodeSet results)

This is the method for initiating a query. Note that it has "Standard" in the name whereas the ones below do not. It does not checking for some qualifying characters so it is better to use FindMatchingSet (5.4.3.8.3)

5.4.3.8.2 int FuncOnt::FindMatchingSet(string packedQuery, NamedNodeSet results)

This method expects a triple in the form of "subject;predicate;object." This method unpacks the triple and then calls FindMatchingSet (5.4.3.8.3).

5.4.3.8.3 int FuncOnt::FindMatchingSet(string Sub, string Pred, string Obj, NamedNodeSet results)

This one will call the standard function but will also check to see if one of the concepts refers a set in the results lists. If any of the triple values have an asterisk (*) as the first character it is considered to indicate the name of a set in the results set. This set is then passed to the appropriate FindMatchingSet method (5.4.3.8.4, 5.4.3.8.5, or 5.4.3.8.6).

5.4.3.8.4 int FuncOnt::FindMatchingSet(LinkedList<BaseOntNode> Sub, string Pred, string Obj, NamedNodeSet results)

Usually called by FindMatchingSet. This version loops through the list of concept nodes sent in as the subject and calls FindMatchingSet (5.4.3.8.3) for each one.

5.4.3.8.5 int FuncOnt::FindMatchingSet(string Sub, LinkedList <BaseOntNode> Pred, string Obj, NamedNodeSet results)

Usually called by FindMatchingSet. This version loops through the list of concept nodes sent in as the predicate and calls FindMatchingSet (5.4.3.8.3) for each one.

5.4.3.8.6 int FuncOnt::FindMatchingSet(string Sub, string Pred, LinkedList <BaseOntNode> Obj, NamedNodeSet results)

Usually called by FindMatchingSet. This version loops through the list of concept nodes sent in as the object and calls FindMatchingSet (5.4.3.8.3) for each one.

5.4.3.9 bool FuncOnt::ReadRdfXml(string filename)

The ReadRdfXml method reads the triples from an RDF XML file. If the RecompileOnDemand flag is false then it will also call the RecompileAll() method. The filename will be used as the current ontology name.

5.4.3.10 void FuncOnt::ReadFromOntologyStore(OntologyStore.OntologyStore theStore)

A library developed as part of the reasoning engine GUI has a namespace and class called "OntologyStore." The ontology store is a memory container that holds triples. The triples in the ontology store can be viewed with a C#/XAML GUI.

5.4.3.11 LinkedList<string> FuncOnt::GetOntologyNameList()

Returns a linked list of all the ontology names. The names were either from RDF/XML files or from the current ontology name being set.

5.4.3.12 bool FuncOnt::WriteOntologyFile(string ontologyName)

Writes out an RDF/XML file of the specified ontology. The file name is the same as the ontology name.

5.4.3.13 bool FuncOnt::RecompileAll()

Takes all the wrapped code and compiles it into a ".dll" file in the temporary directory. The ".dll" file is instantiated in a new application domain and references to the appropriate classes are created for user functions and predicate functions.

5.4.3.14 bool FuncOnt::RunUserFunction(string funcName, NamedNodeSet results)

Calls the Invoke method, with parameters of the "UserMain" method and the ontology class and results class as parameters, of the class created for the user function.

5.5 Summary

There are two main classes that are frequently instantiated or used be code interfacing with the reasoning engine: the reasoning engine class FuncOnt and the answer set class NamedNodeSet. The primary interaction with the reasoning engine will be to call the query method FindMatchingSet(). The NamedNodeSet class is used to hold the results of all the queries and consists of a dictionary class (set member) that holds the answer set lists and then several methods that perform set logic on the lists. The two primary code snippets that area wrapped by the reasoning engine are the fo:PrimaryCode and fo:UserCode. Each of these two methods has defined method parameters which should be used as part of the code.

References

ISO. 2006. Common Language Infrastructure (CLI) Partitions I to IV. Information Technology. http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=58046

CHAPTER 6

SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS

The motivation behind this dissertation was to facilitate a change in hydrologic modeling through the use of knowledge modeling. The intent was to allow us as hydrologic scientist and engineers to focus more on the weightier matters by automating processes that can be standardized. Additionally, formalizing the concepts and relationships between them can have the effect of generating a scientific discussion of parts of hydrology that are more art than science, potentially resulting in new scientific advances. The literature and software review examined tools for modeling hydrologic knowledge with the aim of enhancing the use of existing hydrologic computational tools. The conclusion of the literature and software review is that there was no existing software that could adequately capture both the procedural and semantic aspects of applied hydrology. The result of the research of this dissertation is a new knowledge modeling tool as well as investigations into and demonstrations of its application to hydrologic tasks. The results show that integrated semantic and procedural knowledge modeling is quite complimentary to the tools and tasks of hydrology.

6.1 Summary and Conclusions

The initial research and examination of software, Chapter 1, found that a significant amount of knowledge modeling work has been done in the fields of semantic modeling and model driven software engineering. Because semantic modeling is geared towards utilizing webs of concepts in a very general form, semantic modeling seemed to hold the most promise for capturing the concepts and thought processes we use as hydrologists. Semantic

modeling is still a long way off from capturing and effectively using process descriptions of the sort we frequently use in hydrology. In order to overcome this shortfall and enable the effective use of knowledge models for hydrologic purposes, we investigated and developed a new approach for combining semantic reasoning with procedural knowledge descriptions.

The second chapter details the accomplishments of the research and development into integrating semantic models with procedural knowledge models. The result was the recognition that there was no formal semantic logic to describe the "how to" of a predicate along with the creation of a new, proof-of-concept, reasoning engine and knowledge model description that includes this capability. The knowledge model description is based on existing formal semantic logics. A set of new formal semantic logic terms were created (http://chl.erdc.usace.army.mil/FO-lang-20111201#) to add procedural knowledge to the knowledge base. Two fundamental types of procedural knowledge were chosen for inclusion as part of the formal semantic logic terms – concepts that form a "how-to" for a semantic verb and concepts that form an overarching set of steps similar to a menu function. In addition, procedural knowledge is allowed in the form of helper methods and library-style classes. This additional procedural knowledge is not used by the reasoning engine but only by the two primary forms of procedural knowledge. The form of procedural knowledge model chosen for inclusion is source code.

The semantic and procedural reasoning engine created, termed a functional ontology reasoning engine, has semantic deductive capabilities to answer semantic queries as well as the capability of wrapping, compiling, and executing the source code snippets included in a functional ontology knowledge base. The wrapping of the "how-to" and menu-style source code snippets allows for the source code to have full access to the query mechanisms of the

semantic engine as well as utilize the full set of libraries and inherent functionality of the source code language, in this case C#. User-created libraries can also be included as part of the procedural knowledge. The capabilities of the new reasoning engine and semantic and procedural knowledge base are quite broad and could apply a wide variety of knowledge modeling situations. The goal of the Chapters 3 and 4 were research and demonstration into how the semantic and procedural knowledge base and reasoning engine facilitates our use of hydrologic tools and computational models. The result in both cases is both a set of knowledge modeling patterns that enable complex analysis and a practical example.

The functional ontology knowledge model represents a new paradigm for approaching how we conceptually and practically create and use hydrologic models and computational tools by bringing together source code and semantic concepts so as to enable the two to utilize each other directly. In the current semantic modeling paradigm procedural knowledge, source code, is external to the semantic knowledge base and thus by definition not able to be utilized by the semantic knowledge base or the reasoning engine. The functional ontology paradigm moves the source code from external to the knowledge base to an internal part of the knowledge base. Thus the reasoning engine is able to include very complex procedural knowledge and is able to do so much more than a standard reasoning engine is able to do, such as download data from an external source, run a command-line program, or compute a highly complex algorithm with fall-back measures and error-checking, etc., all the while having each action tied to specific concepts and semantic logic.

The example case presented in Chapter 2 is a simple example meant to illustrate the basic concepts of the functional ontology paradigm. The example uses a procedural description of how to compute areas and perimeters of polygons as part of a simple ontology

of polygons. The results illustrate that 1) the reasoning engine is able to execute procedural knowledge as part of the deductive process, 2) that the procedural knowledge descriptions are able to execute queries on the ontology to obtain needed information, and 3) that the procedural knowledge is able to include error-checking and conditional statements.

Chapter 3 delved into a much more complex application of the semantic and procedural knowledge models, by an order of magnitude (in terms of number of concepts, the number of relationships between concepts, and the number of knowledge model patterns used). The goal of Chapter 3 was to demonstrate the relevance and utility of semantic and procedural models for typical hydrologic tasks, in this case watershed delineation. Many of the computational tools we use as hydrologists have an informal, and somewhat implicit, ontology that for their input and output data and use. This is the case with the TauDEM (Tarboton et al., 2009) suite of tools.

TauDEM is a suite of command-line executable programs that deduce from Digital Elevation Models (DEMs) various other forms of knowledge, such as watershed boundaries and stream locations. Each of the executable functions takes various input data sets and produces one or more output data sets. There are specific flags to denote the meaning of the various files along with a suggested naming convention that acts as an informal semantics. The user of TauDEM needs to learn the many file extension conventions, the command-line flags, and what each of the many TauDEM functions accomplishes. Situations such as this are ripe for the application of semantic and procedural knowledge models.

A semantic and procedural knowledge model of the TauDEM suite of tools was created to automate the delineation of watersheds for a given project purpose. The semantic models covered the types of data, the TauDEM functions, their input and output data set

requirements, the command-line flags for the data sets, as well as the set of project purposes the user is able to choose from and the relationship between those purposes and the TauDEM command-line flags. The procedural knowledge models cover the procedural knowledge for the second-order deductive logic to enable the reasoning engine to deduce the chain of functions required to create the specified data set. Further, the procedural knowledge is able to create the command line to run each of the functions as well as actually execute the TauDEM functions themselves, check for the output files, and add the knowledge about the new data to the ontology. The results illustrate that 1) the informal semantics used by hydrologic tools is able to be represented in a formal manner that enhances the tools we use by facilitating their automated use and execution, 2) that functional ontologies are able to include and use procedural knowledge that greatly extends the analysis capabilities of the reasoning engine, such as executing command-line functions and checking for the existence of files –capabilities that greatly facilitate the real-world application of semantic models, and 3) that semantic and procedural knowledge models are able to capture, deduce, and apply the consequences of the project purpose to the deductive analysis.

The final application of semantic and procedural knowledge models for hydrology, presented in Chapter 4, builds on the work of Chapter 3. Instead of just applying procedural and semantic models to hydrologic tools with similar data and execution requirements, the work of Chapter 4 applies procedural and semantic models to integrate dissimilar computational models. The resulting knowledge model framework allows for the deduction of desired decision data sets from the supplied scenario alternative data sets. This model is an order of magnitude more complex than that of Chapter 3 and represents a real-world

application of semantic and procedural knowledge modeling in a complex analysis situation. The goal of the semantic and procedural knowledge models of Chapter 4 is to create a proof-of-concept framework that is able to integrate computational models into a deductive reasoning framework. The work in Chapter 4 involves both the development of a set of underlying model integration knowledge models as well as instances of semantic models of computational models.

The research into the application of semantic and procedural knowledge models for computational model integration resulted in the creation of several application level functional ontologies, such as those for spatial location, time, scale, and various data types. The research also resulted in the creation of a computational model semantic metamodel. This metamodel approach creates the semantics that describe the overall input, output, and execution semantics in terms that the metamodel procedural knowledge models can use to generically treat the computation model in the deductive logic reasoning process. At the same time, the metamodel approach allows the semantic models of the computational models to create and use all the semantic and procedural knowledge it requires to fulfill the requirements of the metamodel.

Further, the research pointed out the central role of the data sets in the deductive logic process. It isn't the computational models that are used for decisions but the data. The data requirements have to drive the computational modeling process. The data sets description must specify the data type, the extent and internal scale (both time and space) of the data, and the scenario used to create the data. The specification of data sets is also a metamodel. This procedural knowledge for the data set metamodel facilitates the deduction of which data sets can be created from other data sets and the computational models

needed to deduce the required data. Creating these procedural and semantic metamodels of computational models and data creates a powerful deductive workflow engine that is able to execute many complex computational models.

The application for the model integration exercise is a proof-of-concept, for an actual project underway, that semantic and procedural modeling can be used as an integration mechanism. The research identified that a key concept to include and effectively use (in a semantic model integration framework) a variety of computational models and data, and to use the models and data in a project setting, is the use of functional ontology metamodels. Functional ontology metamodels (and meta-metamodels) were created to facilitate the use of computational models, the inclusion of various data, and also the project framework.

The results of the model integration research illustrate that 1) through the use of a modular ontology development approach the semantic and procedural modeling process can scale up to include complex applications, 2) that, through the use functional ontology metamodels, the concepts and operation of very complex computational models can be abstracted and integrated into a deductive analysis framework in order to create a powerful deductive workflow engine, and 3) that a data set functional ontology metamodel definition that defines data scale (time and space extent and internal scale) and the scenario used to create the data set is sufficient to enable the reasoning engine to deduce which data sets can be used to derive the desired data sets; integrating the data set definition with the semantic and metamodels of existing computational models allows the reasoning engine to execute the chain of logic to create the desired data.

The successful computational model execution and model integration results of the hydrologic applications demonstrate that semantic and procedural modeling is a very

complimentary technology for computational modeling. The semantic models are able to describe the wide variety of concepts we use in computational hydrologic modeling. The procedural models are able to compliment and inform the semantics with “how-to” knowledge that enables the computer to reproduce, rapidly and effectively, the “how-to” knowledge we employ. Semantic and procedural models are able to effectively wrap the hydrologic tools and computational models we use in a fashion that enhances and automates their use. Thus semantic and procedural models promise us the ability to automate many tasks that we do now in a manner that integrates into other knowledge as opposed to being a stand-alone process description. The promise of semantic and procedural modeling will allow us to capture existing scientific knowledge in a readily-usable form, enabling greater scientific advances by allowing us to spend more time on researching new scientific knowledge.

Finally, since the reasoning engine is a library that can be included other software, as well as used within the procedural knowledge, instructions on how to use the reasoning engine are needful. Chapter 5 is a user’s manual that covers the reasoning engine functions and how they can be used. It also details the code wrapping for the procedural knowledge so that users can know what method headers to expect and how the code is called as part of the reasoning engine.

6.2 Recommendations

Because this research produced a new knowledge modeling tool, the first recommendation for future work is a practical one, create a new user interface for developing the knowledge models that assists the user in visualizing the knowledge model and debugging the source code. Currently used knowledge modeling interfaces, such as the

Protégé software (Stanford Center for Biomedical Informatics Research, 2010), do not allow for properties of predicates to be included. The functional ontology semantic logic specification is simply a set of potential properties for predicates. Properties of predicates is a rather esoteric feature in the field of semantic modeling but is included in the OWL-FULL (Patel-Schneider et al., 2004) specification. Allowing predicates to have properties breaks the "decidability" property of knowledge models (i.e. knowing that the computer can reach a final result). As such properties of predicates are typically shunned by the semantic modeling community in favor of decidable semantic model specifications such as OWL-Lite or OWL-DL. The typical user interfaces for developing knowledge models, then, do not allow for the types of knowledge models presented in this dissertation.

A second practical area of development will need to focus on sharing the semantic and procedural knowledge models. For the knowledge models to be useful they must first be obtainable. There are several ways currently used to disseminate ontologies or software and any one of a number of these approaches could be adapted for use for disseminating the functional ontology knowledge models. Typically these involve an on-line repository controlled by an institution that creates them (e.g. NASA SWEET Ontologies) or links to ontology files built into web pages (e.g. the Semantic Web).

A third practical area of development should focus on making robust application-level functional ontologies, such as units conversion, multi-dimensional data storage and transformation, and tools to obtain data from on-line data sources such as the CUAHSI HIS (Tarboton et al., 2011). Additionally, a primer for semantic and procedural modeling with an application to wrapping existing computation models would assist in bringing more computational models into the set of models that can be used in model integration.

A next step in semantically-mediated model integration would be to develop the reasoning process for types of tightly coupled model components, such as OpenMI (Gregersen et al., 2007) or the Common Component Architecture(CCA) (CCA, 2004). Instead of executing the individual models it would assemble the model components into the final model using the specified model integration technology.

A potentially significant research area for procedural and semantic modeling for hydrologic applications is the development and formalization of the watershed investigation processes we now use. Given the abundance of available on-line data, the opportunity exists to begin the development of automated model creation processes. These semantic and procedural models should incorporate knowledge about the aspects of watersheds that we consider important for model purposes and the relative importance of the physical processes present in the watershed in the overall modeling process.

The goal of this future work will be to create a framework that follows the three stages of the philosophy of computational modeling. The first stage of the framework will create the perceptual model. Creating the perceptual model will require a capability to integrate general hydrologic modeling principles, the project purpose, and automated data collection and testing tools (see Figure 6-1). Once the perceptual model is created, the second stage of the framework will need to integrate knowledge about how scale affects process requirements is integrated, along with knowledge about conceptual model components, in order to create the “digital” conceptual model (see Figure 6-2). The third stage will begin with the digital conceptual model and the input data and translate the information into a generic model description. This generic model description can then be

turned into the input data for any numerical model that simulates the required conceptual models (see Figure 6-3).

The framework, shown schematically in Figure 6-1, Figure 6-2, and Figure 6-3, will require several different kinds of semantic and procedural knowledge which correspond to the kinds of knowledge a hydrologic modeler must know. There are fundamental concepts and relationships between those concepts (e.g. the hydrologic cycle) that form the theoretical foundation for the modeling work. Next there is practical knowledge, such as the implications of both the goal of the study and the properties of the area to be modeled.

These implications mold the theoretical concepts into a set of practical concepts that should be modeled. This set of concepts that is molded from theoretical into increasingly practical concepts forms a dynamic, evolving body of knowledge. This body of knowledge, specific situational concepts about the model needed, is separate from both the theoretical

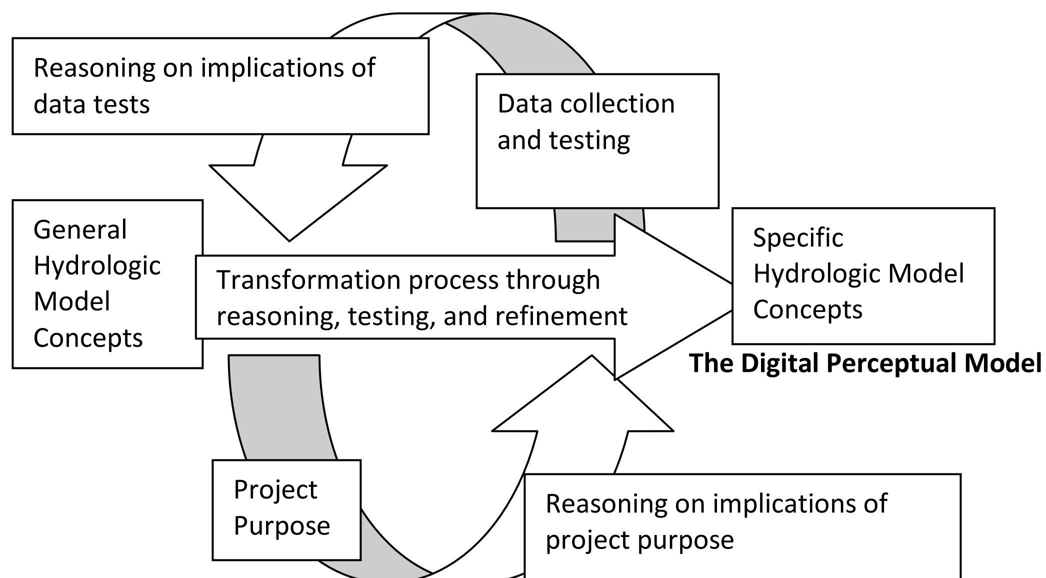


Figure 6-1. The creation of the digital perceptual model from a generalized set of hydrologic model concepts.

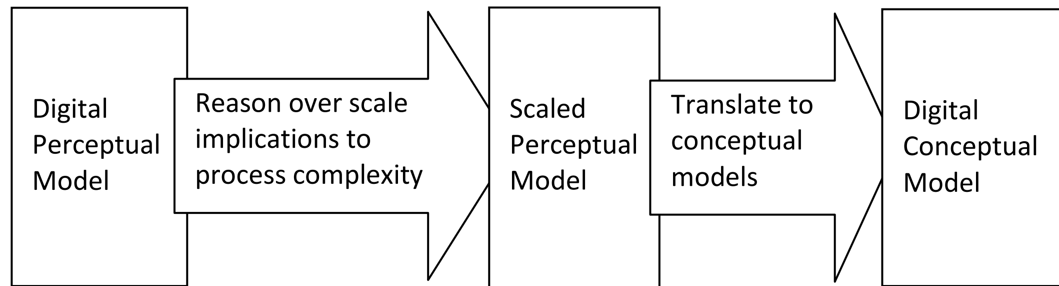


Figure 6-2. The creation of the digital conceptual model from the digital perceptual model.

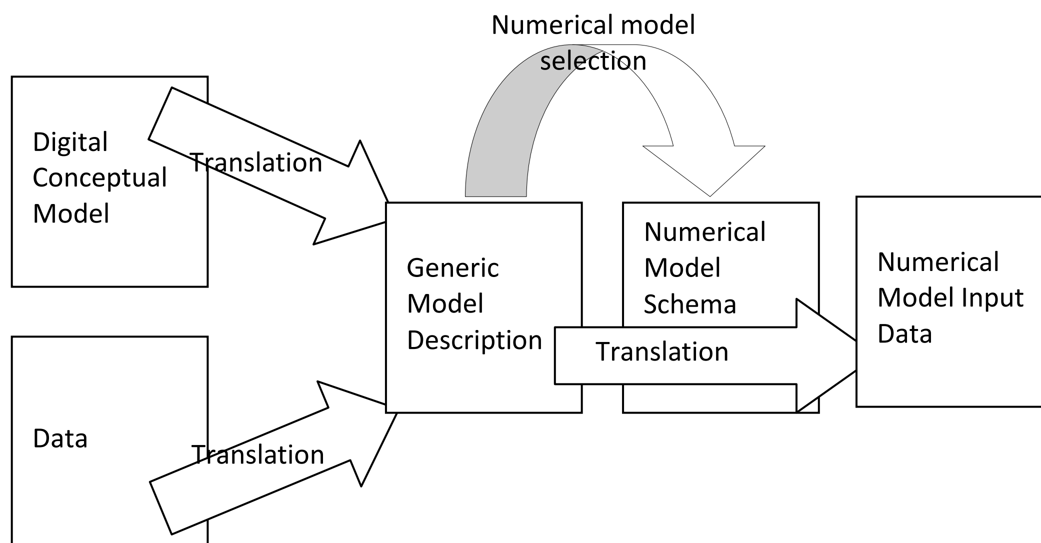


Figure 6-3. The creation of the numerical model input data from the digital conceptual model and data.

and practical general knowledge. A fourth set of knowledge is how to interact with and investigate the world. This involves both an ability to actively operate on and test data about the world as well as utilize the knowledge from the tests to evolve the modeled-situation specific knowledge.

There are also several interesting avenues of research in knowledge modeling. In addition to the potential knowledge discovery and investigation afforded to many physical,

social, and other scientific fields, functional ontologies offer the means of creating tools that reason about what they need to reason about. There is nothing precluding a functional ontology from containing code that uses other functional ontologies. An interesting, and exciting, possibility is to create ontologies that are able to discover, test, and decide what new truths and procedures to accept and add to the repository. Because of their nature in integrating algorithms with assertive statements, functional ontologies can include or use any algorithms, including other artificial intelligence algorithms such as neural networks or support vector machines, which can help in identifying and deciding truths and algorithms for identifying additional truths. The challenge in this process, and indeed in all artificial intelligence applications, is adding the creative spark to the program – programming imagination coupled with abstract pattern recognition in order to search for new underlying truths, investigative procedures, and creative processes.

References

CCA, 2004. Software. Common ComponentArchitecture. The Common Component Architecture Forum. <http://www.cca-forum.org>, [Accessed 2 April, 2012].

Gregersen, J.B., Gijsbers, P.J.A., Westen, S.J.P., 2007. OpenMI: Open Modeling Interface. In: Journal of Hydroinformatics 9(3): 175-191.

Patel-Schneider, P. F., Hayes, P., Horrocks, I. 2004. OWL web ontology language semantics and abstract syntax, W3C. <http://www.w3.org/TR/owl-features/>, [Accessed March 5, 2010]

Stanford Center for Biomedical Informatics Research, 2010. Protégé. <http://protege.stanford.edu/>, [Accessed March 5, 2010].

Tarboton, D.G., Maidment, D., Zaslavsky, I., Ames, D., Goodall, J., Hooper, R.P., Horsburgh, J., Valentine, D., Whiteaker, T., Schreuders, K., 2011. Data Interoperability in the Hydrologic Sciences, The CUAHSI Hydrologic Information System. Proceedings of the Environmental Information Management Conference 2011. M. B. Jones and C. Gries. Santa Barbara, CA, University of California**:** 132-137.

Tarboton, D.G., Schreuders, K.A.T., Watson, D.W., Baker, M.E., 2009. Generalized terrain-based flow analysis of digital elevation models. 18th World IMACS Congress and MODSIM09 International Congress on Modelling and Simulation. R. S. Anderssen, R. D. Braddock and L. T. H. Newham. Cairns, Australia, Modelling and Simulation Society of Australia and New Zealand and International Association for Mathematics and Computers in Simulation: 2000-2006.

CURRICULUM VITAE

Aaron Byrd, 2013

EDUCATION

B.S. Civil and Environmental Engineering, Brigham Young University, 2002. Minor in Geology.

M.S. Civil and Environmental Engineering, Brigham Young University, 2003. Emphasis in Hydrology, Minor in Business Management.

PROFRESSIONAL REGISTRATION

Professional Engineer, State of Mississippi

WORK EXPERIENCE

December 1999 – September 2003: Research Assistant, Environmental Modeling Research Laboratory, Brigham Young University. Developed the user interface for the Gridded Surface Subsurface Hydrologic Analysis (GSSHA) model in the software Watershed Modeling System (WMS.)

October 2003 to December 2011: Research Hydraulic Engineer at the Coastal and Hydraulics Laboratory, Engineer Research and Development Center, U.S. Army Corps of Engineers (USACE-ERDC-CHL). Conduct research in hydrologic model development (GSSHA) and its application for watershed management and engineering, including flooding potential, sediment resource management, water quality management, storm surge modeling, and applications to ecological modeling. Conducted studies in New Mexico, California, Illinois,

Louisiana, Texas, South Carolina, Wisconsin, Minnesota, Alaska, Guam, Georgia, and New York. Developed and provide training for the use of the GSSHA model. Mentored intern and early career engineers. Lead research and applications projects totaling roughly $5M. Assisted the U.S. Department of State for international water resources evaluation and recommendations.

January 2012 to Present: Chief, Hydrologic Systems Branch, USACE-ERDC-CHL. Oversee the work of 15 employees. Provide guidance to early career employees and assist senior engineers in managing projects. Recruit and hire new employees.

AWARDS

ERDC Research and Development Award, 2008, Development of the Nutrient Sub Module Water Quality Model.

ERDC Outstanding Team Effort Award, 2005, Simulation of Land Use Evolution and Toxics Transport.

Civilian Achievement Award, 2007, IPET Report Development (Hurricane Katrina analysis.)

Commander’s Award for Civilian Service, 2013, Hydrology Team Leader supporting inland flood modeling for Hurricane Irene, Superstorm Sandy, and the following Nor’easter.