ABSTRACT

Title of thesis: **Distributed System Behavior Modeling of Urban Systems with Ontologies, Rules and Message Passing Mechanisms**

Maria Coelho, Master of Science, 2017

Thesis directed by: Associate Professor Mark Austin
Department of Civil and Environmental Engineering and ISR

Modern infrastructures are defined by spatially distributed network structures, concurrent subsystem-level behaviors, distributed control and decision making, and interdependencies among subsystems that are not always well understood. This work presents a model of system-level interactions that simulates distributed system behaviors through the use of ontologies, rules checking, and message passing mechanisms. We take initial steps toward the behavior modeling of large-scale urban networks as collections of networks that interact via many-to-many association relationships. We conclude with ideas for scaling up the simulations with mediators assembled from Apache Camel technology.

Last Modified: April 20, 2017

Distributed System Behavior Modeling of Urban Systems with Ontologies, Rules and Message Passing Mechanisms

by

Maria Coelho

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2017

Advisory Committee:
Associate Professor Mark Austin, Chair/Advisor,
Associate Professor Kaye Brubaker,
Professor Bilal Ayyub.

Acknowledgments

I would first like to thank my thesis advisor Dr. Mark Austin of the Department of Civil and Environmental Engineering at the University of Maryland. The door to Dr. Austin's office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right the direction whenever he thought I needed it. I could not have imagined having a better advisor and mentor.

I would also like to thank Dr. Bilal Ayyub and Dr. Kaye Brubaker for serving on my thesis committee, and for their valuable input.

Thank you to my fellow graduate students, Parastoo Delgoshaei and Leonard Petanga, for your help, advice and support through my studies.

Finally, I must express my very profound gratitude to my parents Rachel and Luiz, and to my two brothers Mateus and Joao, for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Table of Contents

List of Abbreviations

API	Application Programming Interface
CPS	Cyber-Physical Systems
DL	Description Logics
DSO	Domain-Specific Ontologies
GIS	Geographic Information System
GML	Geographical Markup Language
GUI	Graphical User Interfaces
IRI	Internationalized Resource Identifiers
ISO	International Organization of Standardization
JAXB	XML Binding for Java
JTS	Java Topology Suite
MBSE	Model-Based Systems Engineering
OOD	Object Oriented Design
OSM	Open Street Map
OWL	Web Ontology Language
RCC	Region Connectedness Calculus
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
SPARQL	Simple Protocol and RDF Query Language
SysML	System Modeling Language
UML	Unified Modeling Language
U.S.	United States
W3C	World Wide Web Consortium
WWW	World Wide Web
XML	Extensible Mark-up Language

Glossary of Terms

This glossary provides definitions of key terms employed in this work:

Action: (Effect) is the response given to stimuli in a transition, and will normally corresponds to an activity performed during the transition in the statechart.

API: (Application program interface) is a set of routines, protocols, and tools for building software applications. An API specifies how software components should interact.

Association: An association represents a linkage (i.e. a connection line) between two classes in an ontology or a class diagram. An association can have a name can be adorned with role names, ownership indicators, multiplicity, visibility, and other properties. Bi-directional and uni-directional associations are the most common types of associations.

Agent: An independently operating Internet program, typically one that performs background tasks such as information retrieval or processing on behalf of a user or other program.

Bi-directional Association: Refers to a symmetric dependency between two classes.

Block Diagram: A SysML diagram, the represents the principal components of a system and the structural design that connects them together.

Cardinality: In the context of databases, cardinality refers to the uniqueness of data values.

Class Diagram: A UML diagram, which focuses on different classes of the software systems and their connection with respect to each other.

Constraint: A design constraint refers to some limitation on the conditions under which a system is developed.

Controller: (Mediator) A component of MVC design pattern, that acts as a communication channel between the model and the view.

Description logic: (DL) is a family of logic-based knowledge representation languages that can be used to represent the terminological knowledge of an application domain in a structured way.

DogOnt: DogOnt is an ontology model designed for supporting interoperation, integration and intelligence in domotic environments.

Domotics:(DOMus infOrmaTICS) Information technology in the home.

Event: Stimuli that may cause a transition from one state to another state in statechart. There are four main categories of events: Signal, time, change and call events.

Extended Markup Language (XML): The extensible Markup Language provides the fundamental layer for representation and management of data on the Web.

First-order logic (FOL): symbolized reasoning in which each sentence, or statement, is broken down into a subject and a predicate. The predicate modifies or defines the properties of the subject. In first-order logic, a predicate can only refer to a single subject.

Individual: Is a semantic web terminology that represents an instance of a class in the ontology.

JAXB: XML binding for Java.

Jena: Jena is an open source Java framework for building Semantic Web and linked data applications.

Jena Rules: Jena Rules is an inference (reasoning) engine that plugs into Jena.

Listener: (Observer) A class that registers its interest to be notified for changes in other classes (Observable) in observer design pattern.

Map: a diagrammatic representation of an area of land or sea showing physical features, cities, roads, etc.

Mediator: In mediator pattern, a mediator, defines an object that encapsulates how a set of objects should interact.

Mediator Pattern: A behavioral design pattern that is used to manage algorithms, relationships and responsibilities between objects. It mitigates the need for point-to-point connections between objects by defining an object

that controls how a set of objects will interact. Loose coupling between colleague objects is achieved by having colleagues communicate with the mediator, rather than with one another.

Model: A model is an approximation, representation, or idealization of selected aspects of the structure, behavior, operation, or other characteristics of a real-world process, concept, or system (IEEE 610.12-1990)

Model-Based Systems Engineering: Model-based systems engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases (INCOSE-TP-2004-004-02, Version 2.03, September 2007).

Model-View-Controller (MVC): Is a system design pattern that separates the representation of information from the user's interaction with it.

Observer Pattern: The observer pattern is applicable to problems where a message sender (observable) needs to broadcast a message to one or more receivers (or observers), but is not interested in a response or feedback from the observers.

Ontology: A model that describes what entities exist in a design domain, and how such entities are related.

Ontology Class: A placeholder for an entity in the system design. An ontology class may have some dataType or objectType properties.

Ontology Instance: An ontology instance is a specific realization of any ontology class object. An object may be varied in a number of ways. Each realized variation of that object is an instance. The creation of a realized instance is called instantiation.

DataType Property: DataType Property defines the relation between instances of classes and literal values, i.e., String using the Protg tool.

ObjectType Property: ObjectType Property defines the relation between instances (individuals) of two classes in semantic web terminology using protg tool.

Ontology Web Language: The Web Ontology Language (OWL) is a knowledge representation languages for defining ontologies.

OptaPlanner: OptaPlanner is a constraint satisfaction solver. It optimizes business resource planning use cases, such as Vehicle Routing, Employee Rostering, Cloud Optimization, Job Scheduling, Bin Packing and many more.

Point-in-polygon: In computational geometry, the point-in-polygon (PIP) problem asks whether a given point in the plane lies inside, outside, or on the boundary of a polygon.

Reasoner (Rule Engine): A semantic reasoner, reasoning engine, rules engine, or simply a reasoner, is a piece of software able to infer logical consequences from a set of asserted facts or axioms.

Reasoning: To infer new statements based on set of asserted facts in the ontology.

Resource Description Framework (RDF): a model for encoding semantic relationships between items of data so that these relationships can be interpreted computationally.

Rule Checking: A mechanism that ensures existing data in the ontology is consistent with rules defined over the ontology. A rule engine often performs this task.

Semantic Web: Refers to W3Cs vision of the Web of linked data.

Semantic Web Layer Cake: An informal term used to describe the stack of technologies used in the implementation of the Semantic Web.

Semantic Web Technologies: Semantic Web technologies provide features to build vocabularies, and develop rule repositories and ontologies.

Software Design Patterns: In software engineering a design pattern is a general reusable solution description to a recurring problem.

SysML: The Systems Modeling Language (SysML) is a graphical modeling language used to define models of systems structure and system behavior.

Transition: A transition is a set of actions to be executed when a condition is fulfilled or when an event is received.

Transitivity: State of being or relating to a relation with the property that if the relation holds between a first element and a second and between the second

element and a third, it holds between the first and third elements. For example, equality is a transitive relation.

Unified Modeling Language: UML is a graphical modeling language used to define mainly software systems structure and behavior.

View: Visual representation of the model in MVC design architecture.

Zone: an area or stretch of land having a particular characteristic, purpose, or use, or subject to particular restrictions.

List of Figures

Chapter 1: **Engineering Urban Systems with Semantic Models**

1.1 Problem Statement

1.1.1 Modern Urban Infrastructure Systems

The modern way of life is enabled by remarkable advances in technology (e.g., the Internet, smart mobile devices, cloud computing) and the development of urban systems (e.g., transportation, electric power, wastewater facilities and water supply networks, among others) whose operations and interactions have superior levels of performance, extended functionality and good economics. While end-users applaud the benefits that these technological advances afford, model-based systems engineers are faced with a multitude of new design challenges that can be traced to the presence of heterogeneous content (multiple disciplines), network structures that are spatial, multi-layer, interwoven and dynamic, and behaviors that are distributed and concurrent.

In a decentralized system structure, no decision maker knows all of the information known to all of the other decision makers, yet as a group, they must cooperate to achieve system-wide objectives. Communication and information exchange are

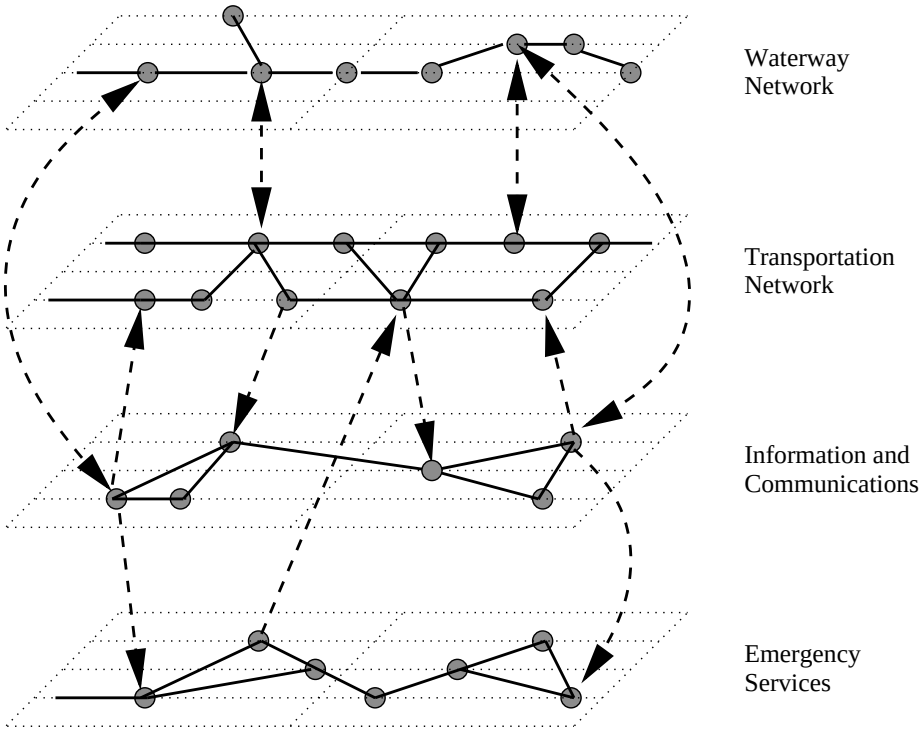

Figure 1.1: Schematic of interdependencies among urban networks.

important to the decision makers because it establishes common knowledge among them which, in turn, enhances their ability to make decisions appropriate to their understanding, or situational awareness, of the system state, its goals and objectives. While each of the participating disciplines may have a preference toward operating their domain as independently as possible from the other disciplines, achieving target levels of performance and correctness of functionality nearly always requires that disciplines coordinate activities at key points in the system operation.

As a case in point, modern urban infrastructure systems comprise physical, communication and social networks that are spatially distributed, and defined by concurrent subsystem-level behaviors, distributed control and decision making, and interdependencies among subsystems that are not always well understood. A typical

setup of urban networks and their dependency relationships. is shown in Figure 1.1.

Engineers keep these difficulties in check by designing subsystems that allow for operational independence, but coordinate at key points to maintain safety and/or handle abnormal circumstances.

Figure 1.2: Concurrent behaviors and interdependent system interactions at intersection of Campus Drive and Baltimore Ave, University of Maryland, College Park, MD [15].

Figure 1.2 shows, for example, cars, pedestrians, and traffic lights at a busy intersection near the University of Maryland. Each of the participating entities favors operational autonomy. However, in order for the overall system to operate efficiently and prevent accidents, drivers watch the traffic lights for instructions, and actions are coordinated in a manner defined by sets of rules (e.g., traffic rules, pedestrian rules).

1.1.2 Cascading Failures in Urban Systems

When cross-domain relationships in urban systems are only weakly linked, they are nonetheless, still linked. When part of a system fails, there exists a possibility that the failure will cascade across interdisciplinary boundaries to other correlative infrastructures, and sometimes even back to the originated source, thus making highly connected systems more fragile to various kinds of disturbances than their independent counterparts. Such outcomes put engineering designers, disaster-relief personnel and urban planners (i.e., decision makers who are responsible for the engineering of recovery processes) in a tough spot where quantitative decision-making regarding the adequacy of urban infrastructure is complicated by the presence of newfound system interactions, incomplete knowledge of the system state, and breakdowns of communication among urban networks.

Experience over the past decade with major infrastructure disruptions, such as the the 2003 Northeast blackout, Hurricane Katrina in 2005, Hurricane Irene in 2011,and 2011 San Diego blackout, has shown that the greatest losses from disruptive events may be distant from where damages started. For example, Hurricane Katrina disrupted oil terminal operations in southern Louisiana, not because of direct damage to port facilities, but because workers could not reach work locations through surface transportation routes and could not be housed locally because of disruption to potable water supplies, housing, and food shipments [27]. To complicate matters, until very recently infrastructure management systems did not allow a manager of one system to access the operations and conditions of another sys-

tem. Therefore, emergency managers would fail to recognize this interdependence of infrastructures in responding to an incident, a fact recognized by The National Strategy for the Physical Protection of Critical Infrastructures and Key Assets [36]. In such situations, where there is no information exchange between interdependent systems, interdependencies can lead to cascading disruptions throughout the entire system in unexpected, undesirable and costly ways.

1.1.3 Project Objectives

The long-term objectives of this project are to explore opportunities for overcoming these limitations with a "city operating system" that monitors environmental and urban processes, and then plans actions to either mitigate the effects of an impending environmental attack and/or recover from damage caused by such events. Figure 1.3 shows the participating entities and their interactions.

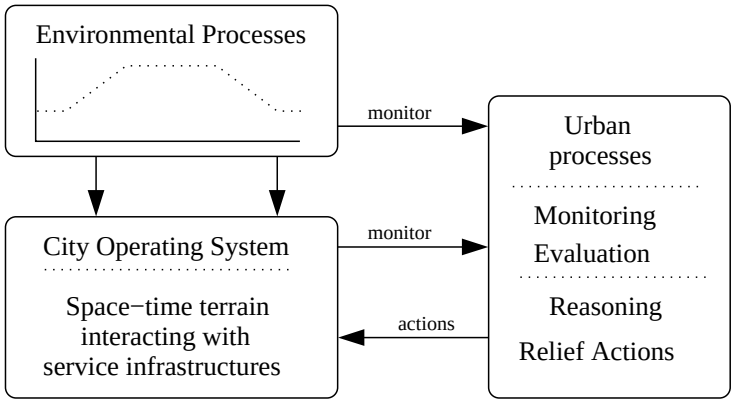

Figure 1.3: Schematic for a city operating system.

This study focuses on a prerequisite to creating this capability, namely, an ability to model the behavior of city-domain processes, and interactions among the distributed

system behaviors within a city.

We envision such a system having an architecture along the lines shown in Figure 1.4, with tools such as OptaPlanner [29] providing strategies for real-time control of behaviors, assessment of domain resilience and planning of recovery actions in response to severe events. Instead of modeling the dynamic behavior of systems with centralized control and one large catch-all network, the work explores opportunities for modeling systems as collections of discipline-specific (or community) networks that will dynamically evolve in response to events. Individual urban domains will operate as concurrent processes each having their own thread of execution, and will respond to streams of incoming data from external domains. Each community will have a graph that evolves according to a set of community-specific rules, and subject to satisfaction of constraints. Communities will interact when then need to in order to achieve system-level objectives. If goals are in conflict, or resources are insufficient, then negotiation will need to take place. Ontologies and rules in the temporal and spatial domains – relief activities need to occur in the right place and the right time – will be integrated with domain-specific ontologies and rules, and support reasoning for simulation and rule-based control.

Ideally this tool will also decision makers to understand how a failure in one network will impact other networks, and what parts of a system are most vulnerable to informed/uniformed attack. It should also allow decision makers to assess the sensitivity of systems to model parameter choices, the influence of resource constraints, and potential emergent interactions among systems.

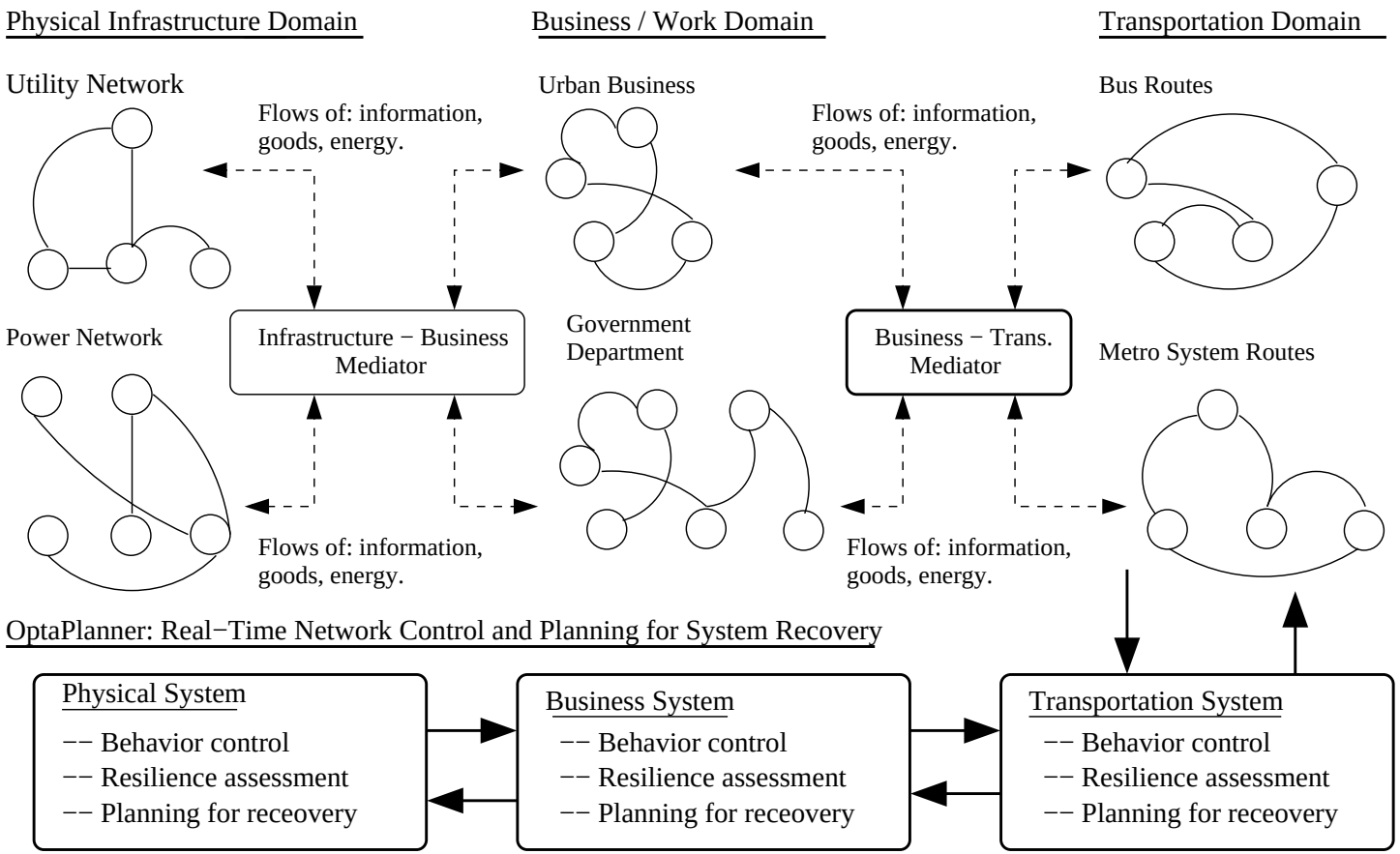

Figure 1.4: Architecture for multi-domain behavior modeling with many-to-many associations. We envision tools such as OptaPlanner [29] providing strategies for real-time control of behaviors, assessment of domain resilience and planning of recover actions in response to severe events.

1.2 Solution Approach

1.2.1 System of Systems Perspective

Definition. Solutions to this project objective are complicated by the fact that cities are system of systems, and not just systems. In "The Art of Systems Architecting" Maier and Rechtin [24] define a system of systems as one in which its components:

1. Fulfill valid purposes in their own right, and continue to operate to fulfill those purposes if disassembled from the overall system, and
2. Are managed (at least in part) for their own purposes rather than the purposes of the whole; the components systems are separately acquired and integrated but maintain a continuing operational existence independent of the collaborative system.

Two key characteristics which derive from this definition are [33]:

1. **Emergence:** Properties which do not belong to any of the constituent parts will emerge from the combined system of systems.
2. **Evolution:** The system of systems will change over time as constituent systems are replaced.

System of Systems Perspective. Notions of emergence and evolution (managed evolution) are central to city development and operation. Consider the following

points:

1. Cities are not conceived or built by an individual organization and are not merely a collection of buildings.

2. A city "emerges and changes over time" through the loosely coordinated and regulated action of individuals to satisfy the needs of its citizenry.

3. City system behaviors are defined by a large, but finite, number of agents. Various agents – people, communities, organizations – build certain parts of cities to satisfy their respective objectives.

4. Cities "grow and flourish" based on societal and economic stimulus, and "falter and fall" into decay when such stimulus is absent.

5. Traditional models of individual agent behavior are defined by relatively simple (deterministic) rules that connect information and resources to action, and are subject to satisfaction of dependency relationships (see Figure 1.1) and urban regulations.

While each of the city subsystems may have a preference to operating as independently as possible from the other subsystems as possible, strategic collaboration among subsystems is often needed to either avoid cascading failures across systems and/or recover from a loss of functionality. Collaboration among subsystems can also result in new services – services that the participating systems cannot achieve by themselves.

Figure 1.5: Composition of ground and air transportation services.

These services need not be complicated. Figure 1.5 shows, for example, an airport acting as an interface between cooperating surface and air transportation services. The key challenge in creating an efficient system is synchronization timetables and capacities so that passengers can transition from one mode of transportation to another.

Evolution of City Fabric. The complexity of cities becomes apparent when we consider the dependencies among agents, information, and resources, and the influence that new technologies can have on the spatial and temporal properties of city fabric. Industrial-age cities tend to be spatially compact – they aim to overcome inefficiencies in communication and transportation by minimizing space to conquer time constraints. Information-age cities work in exactly the opposite way – they employ highly efficient communication networks to minimize the importance of time constraints and relieve the need for urban congestion (relaxed space constraints).

Model-based Systems Engineering (MBSE) for Cities. We believe that the difficulty in overcoming these challenges can be mitigated through the systematic application of model-based systems engineering (MBSE) procedures. MBSE procedures help engineers develop models for products that typically follow a design-build-operate-retire lifecycle [2]. At the front end of system development, use of semi-formal languages, such as SysML [13], helps engineers systematically consider

scenarios for required system functionality, create visual representations (diagrams) for fragments of behavior, develop requirements (constraints) for system performance and economics, and generate design alternatives that have the potential for delivering good design solutions [2]. For a detailed discussion, see Appendix A.

Even though a city does not retire, over time, properties of the city will evolve as elements of the city age and are replaced. MBSE can play a central role in the replacement process. Where state-of-the-art MBSE procedures fall short is in the systematic consideration of interactions among many concurrent behaviors. Visual languages such as SysML [13] are not designed to handle this class of problems.

1.2.2 Ontologies, Rules, and Reasoning Mechanisms

Figure 1.6 presents a framework for the implementation of semantic models using ontologies, rules, and reasoning mechanisms [9]). An ontology is "a set of knowledge terms, including the vocabulary, the semantic interconnections, and some simple rules of inference and logic for some particular topic [16]." To provide a formal conceptualization within a particular domain, and thereby facilitate communication among people and machines, ontologies need to accomplish three things:

1. Provide a semantic representation of each entity and its relationships to other entities;

2. Provide constraints and rules that permit reasoning within the ontology, and

3. Describes behavior associated with stated or inferred facts.

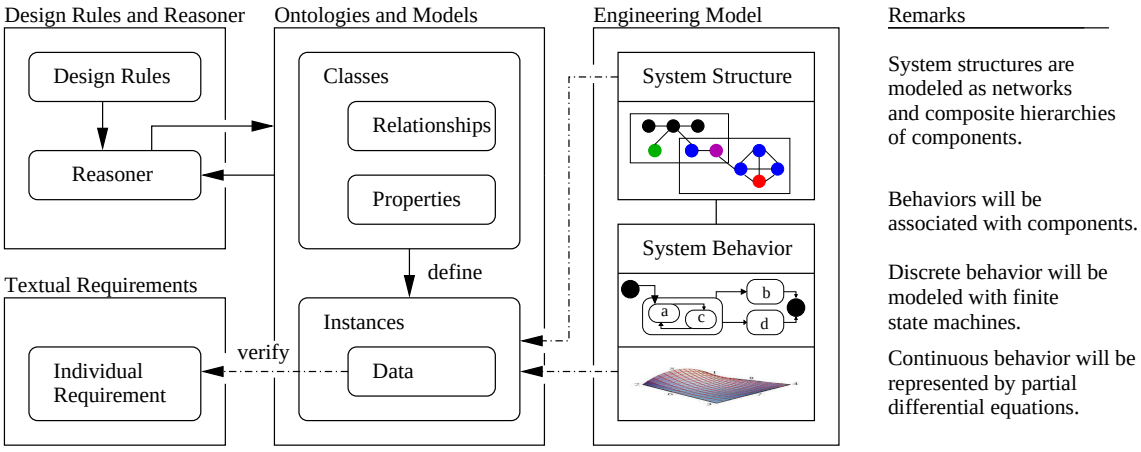

Figure 1.6: Framework for implementation of semantic models using ontologies, rules, and reasoning mechanisms (Adapted from Delgoshaei, Austin and Nguyen [9]).

On the left-hand side of Figure 1.6, textual requirements are defined in terms of mathematical and logical rule expressions for design rule checking. Engineering models of urban system structure will consist of networks and hierarchies of connected components formally described in terms of geometry (e.g., position, size) and connectivity (e.g., connected, touches, disjoint), possibly organized into layers (e.g., a hierarchy of networks). Engineering models of urban system behavior will be combinations of discrete (e.g., statecharts) and continuous (e.g., differential equations) behaviors. The semantic counterpart of engineering models is ontologies (class hierarchies), individuals (graphs), and rules [10, 9], Data contained within the engineering models will be ingested into the semantic model as data property values.

Computation with rules provides several advantages [23, 31]:

1. Rules that represent policies are easily communicated and understood,

2. Rules retain a higher level of independence than logic embedded in systems,

3. Rules separate knowledge from its implementation logic, and

4. Rules can be changed without changing source code or underlying model.

A rule-based approach to problem solving is particularly beneficial when the application logic is dynamic, and where rules are imposed on the system by external entities. Both of these conditions apply to the design and management of urban systems.

1.2.3 Semantic Models of Urban Structure and Behavior

Figures 1.7 and 1.8 build upon Figure 1.2, and show the pathway from observation of concurrent behaviors and interdependent system interactions at a traffic intersection to semantic analysis and event-based modeling of behaviors.

Traditional approaches to the modeling of urban structure and behavior simplify the problem by organizing abstractions into layers that are mapped onto a Cartesian – (x,y) or (latitude,longitude) – coordinate system. Each layer will show a spatial distribution of a property of the city (e.g., land use, population density, average house price) and will have a predefined syntax and semantics for the associated content. Decision analysis procedures are simplified by ignoring dependency relationships among the layers (see Figure 1.1), and by assuming that estimates of system performance can be obtained through the summation of data extracted from select layers.

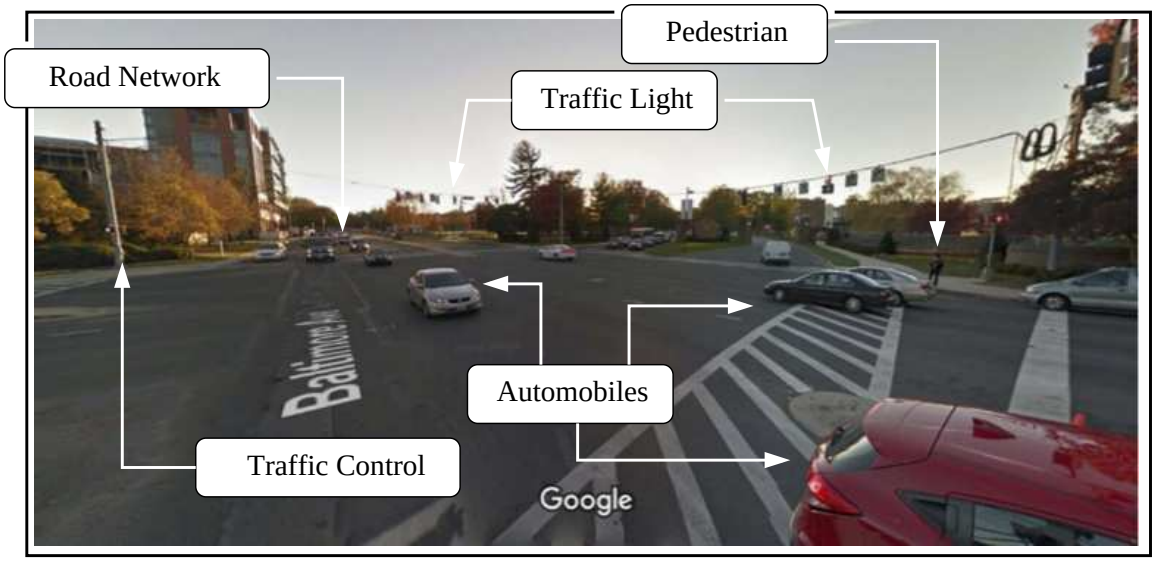

Figure 1.7: Annotation of structure and behaviors at intersection of Campus Drive and Baltimore Ave, University of Maryland, College Park, MD [15].

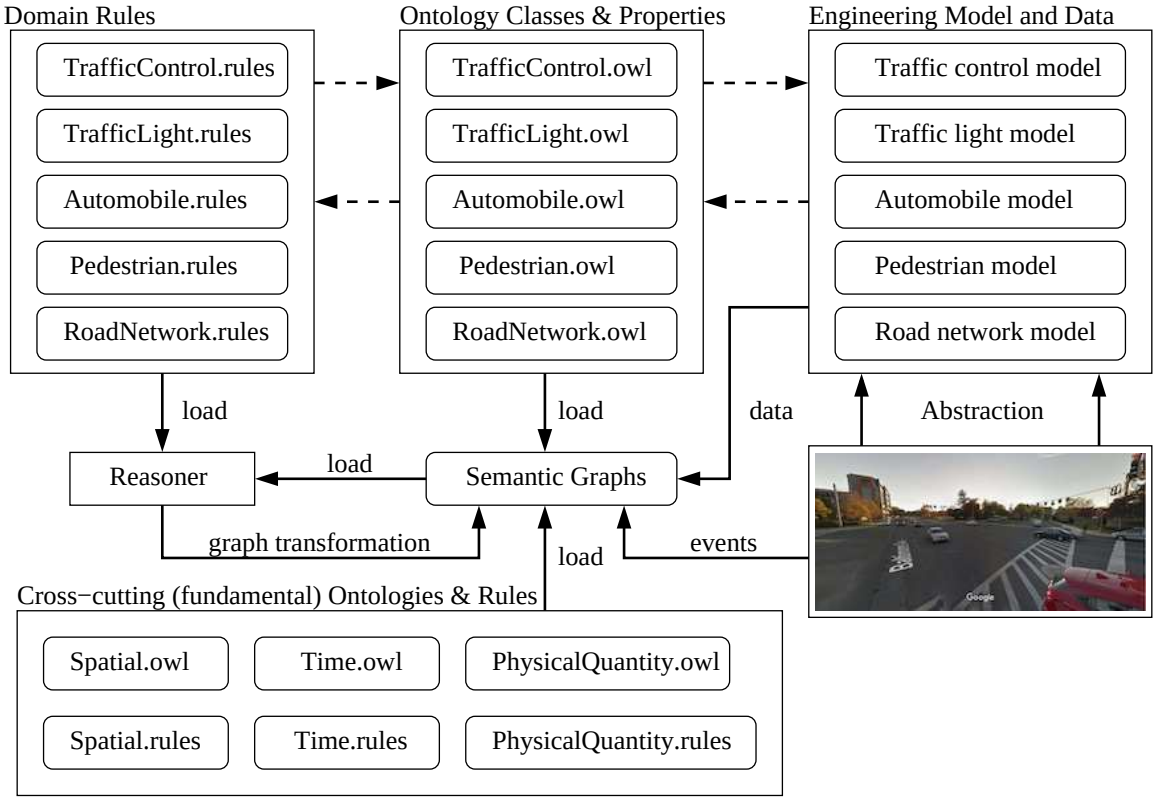

Figure 1.8: Semantic model for traffic intersection.

As cities transition from an industrial- to information-age capability, a more appropriate view is one of networks of networks, where physical, communication, economic, and social processes are intertwined and dynamic. Figure 1.7 builds upon Figure 1.2 by identifying various kinds of objects that participate in the urban scene. Each type of object – automobile, traffic light, pedestrian – will have its own goals (purpose), structure (geometry), and behaviors that are constrained by physics and strategies of control.

Figure 1.8 places the entities identified in Figure 1.7 in a formal setting for analysis. Each type of domain-specific entity will have an ontology and a set of rules (e.g., TrafficLight.owl and TrafficLight.rules), and will be represented within the semantic model as a graph. Some phenomena, such as space, time and physical units, cut across to all domains and are fundamental to understanding whether a decision will take place at the right time and the right place. The actual data for a specific type of traffic light (e.g., position, timing, phase cycle) will come from an engineering model of the traffic light. The geometry of an actual intersection will be imported from sources such as OpenStreetMap.

A key benefit of the proposed approach is that no constraints are placed on the types of relationships that a semantic graph may form. While data properties store the actual data relevant to an individual, object properties represent relationships among individuals that might not even be in the same domain. Engineers can exploit this freedom in the concurrent design of domain specific ontologies and rules. A modeler might find, for example, that if a relationship exists between a traffic light object and the road intersection model (e.g., traffic light X is located at intersection

Y), then this knowledge will simplify the development of rules for traffic control.

1.3 Related Work

1.3.1 Glassbox Simulation Engine

Glassbox [39] is a general purpose data-driven simulation engine created for Maxis games, the most famous being SimCity. The design goal for Glassbox is to provide modelers with an environment where you simulate very simple objects that can be easily composed together to do very complex things.

Every single entity in the city has a distinct simulation. Complex behaviors emerge out of their interactions. As illustrated in Figures 1.9 and 1.10, cities are modeled as resources + units + maps + globals, combined with collections of rules, all packaged into a box. Resources are the basic currency of the simulation (e.g., oil, water, wood). Units represent things like houses and factories. The state of a unit is defined by the collection of resources it owns. Units are also defined by a spatial extent, which in turn, defines its simulation footprint. Maps represent resources in the environment (e.g., coal, oil, a forest). A unit interacts with map through its footprint. Rules are defined by nouns and verbs, and they operate on resources. For example, a rule might move a resource from one place to another, or convert a resource into a new form. Rules also have inputs and outputs, and can handle dependency relationships. A rule can state, for example, that money can be converted into wood, if a person is available. Finally, rules can operate over a variety of target domains (e.g, locally, over a map, or globally) and they can be

Figure 1.9: Glassbox is a data-driven simulation engine for Maxis games, the most famous being SimCity. Cities are modeled as resources + units + maps + globals, combined with collections of rules, all in a box.

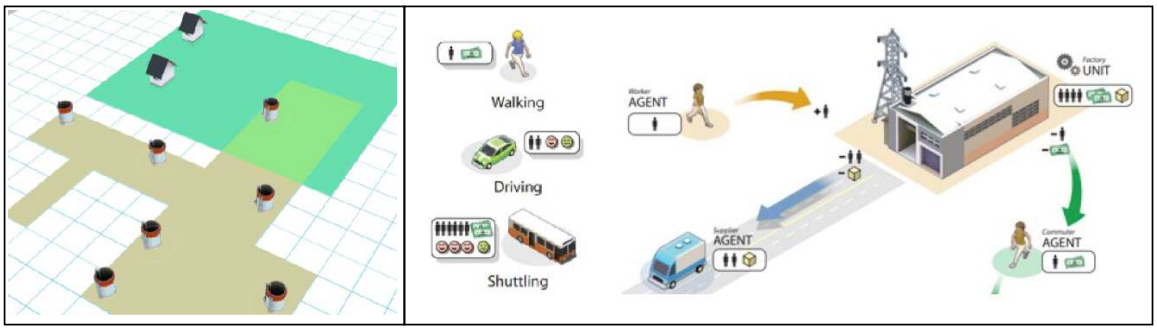

Figure 1.10: Use of zones and agents in the Glassbox Simulation Engine.

chained together. Simulation games are created by defining a play area (SimCity is a game), a variety of unit and map types, and collections of rule scripts. Because units contain their own simulation logic, behaviors can be easily swapped in and out of a game and units can be combined to produce aggregate behavior. More sophisticated behaviors are supported with paths, zones and agents. Paths are points connected by segments; they are used to represent roadways, power lines, water pipes, and flight paths. Zones cover a well-defined area. Rules can be extended to include logic that depends on whether or not a unit is spatially located within a zone. Agents (e.g., cars, trains, pedestrians) carry resources from one unit to another. They are created by unit rules and have a destination (i.e., they are going somewhere to do something). Emergent behaviors, such as traffic jams, are the result of patterns that individual cars (agents) navigate roads (paths). The game engine provides builtin support for basic physics (e.g, water flows downhill).

1.3.2 Urban Domain Modeling with Graphs and Cellular Automata

Numerous researchers have studied the topology of urban environments from a graph theoretic standpoint. For example, Whiting [38] has proposed a method for constructing large-scale graph models of three-dimensional urban content (e.g, roads, walkways, green space) and topology (e.g, adjacency relations) from real-world geometry data. These graph data structures are the basis upon which algorithms can be developed for route finding services.

Our approach to distributed behavior modeling is similar to studies that cap-

ture the temporal dynamics of cities with cellular automata, agent-based models, and fractals [5]. A cellular automata is composed of: (1) a discrete cell space, together with, (2) a set of possible cell states, and (3) a set of transition rules that determine the state of each cell as a function of the states of all cells within, (4) a defined cell-space neighborhood of the cell. Time is discrete and all cell states are updated simultaneously at each iteration [37]. Cellular automata can be adapted to provide both low- and high-resolution views of spatial dynamics and to understand dynamic interactions among the various layered systems (e.g., population density, land use patterns, transportation networks) and flows and consumption of resources (e.g., energy) [6].

1.3.3 Urban Domain Modeling with Ontologies

A detailed discussion the use of ontologies in urban development projects can be found in Falquet, Metral, Teller and Tweed [11]. Ontologies have been developed for the geographic information sector, to model interconnections (mediators) among urban models, and to describe urban mobility processes. Extensive studies have been conducted on the development of ontologies for the geography markup language (GML) and CityML, the XML markup language for cities.

As part of the recent interest in Smart Cities, researchers have proposed so-called smart city ontologies. A close examination reveals that they contain an exhaustive list of things you might find in a smart city, and proposals for relationships among things, but are otherwise not smart at all.

Our viewpoint is that ontologies – classes, and their associated data and object properties – need to be developed alongside rules. A notable effort in this direction is the DogOnt ontology and rules for statechart behavior modeling of devices in home automation [8].

1.4 Contributions and Organization

The contributions of this study are as follows:

1. We provide a framework for modeling concurrent, directed communication between all entities composing a system. The architecture builds upon the framework presented by Austin et al. [3], and in particular, extends the distributed behavior modeling capability from one-to-one association relationships among communities to many-to-many association relationships among networked communities. As illustrated in Figure 1.11, one-to-one association relationships can be modeled with exchange of messages in a point-to-point communication setup. Many-to-many association relationship among systems are enabled by collections of mediators. Each ontology is paired with an interface for communication and information exchange with other ontologies, and hosts a set of domain specific rules as well as the system interaction rules.

2. We employ a novel use of software design patterns. Mediator design pattern is implemented to allow communication management in a system, and visitor design pattern is implemented to allow for data retrieval.

3. We explore mechanisms for incorporating notions of space and time in the reasoning process.

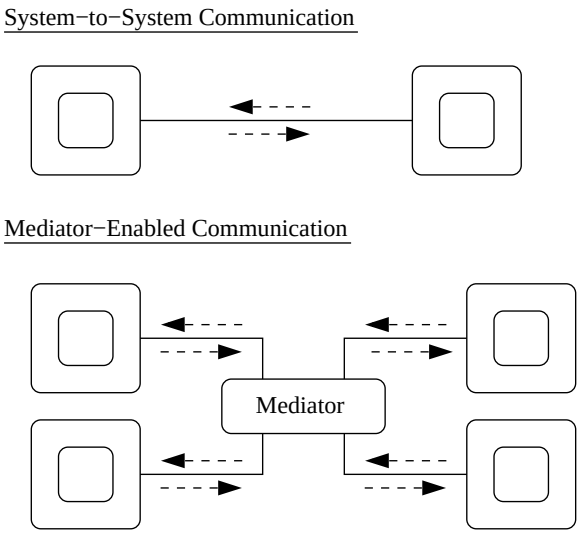

Figure 1.11: Framework for communication among systems of type A and B. Top: point-to-point communication in a one-to-one association relationship between systems. Bottom: mediator enabled communication in a many-to-many association relationship among systems.

There are a number of similarities between the abstractions proposed in our work and those found in the Glassbox Simulation Engine. Glassbox paths and zones are equivalent to ways and multi-polygons imported from OpenStreetMap. Glassbox agents are analogous to messages passed between domains in our work. Experienced gamers complain that sometimes SimCity allows for behaviors that simply would not happen in the real world. The possibility of an under-constrained world is a problem we will face in our work as well.

The thesis is organized as follows: Chapter 2 provides a background on ontologies and rules, explains their relationship to our related work in model-based systems engineering, and explains the concept of Semantic Web. Chapter 3 de-

scribes several aspects of distributed system behavior modeling with ontologies and rules, including: the semantic models, the system architecture, and the use of mediators for behavior modeling of distributed systems having many-to-many association relationships among connected networks. Chapter 4 provides experimental platforms for assembling ensembles of community graphs and simulating their discrete, event-based interactions, and exercise this capability with an application involving collections of families interacting with multiple school systems. Chapter 5 provides a summary and conclusion of the work presented, and proposes ideas for scaling up the simulations in future work with mediators assembled from Apache Camel technology.

Chapter 2: **The Semantic Web**

2.1 Introduction to Semantic Web

This chapter introduces the Semantic Web vision, and the range of technologies found in its implementation. Basic capabilities of the resource description framework (RDF) and Web Ontology Language (OWL) are described. A simple case study problem involving behavior modeling of family dynamics with ontologies (Jena) and rules (Jena Rules) is presented. Once the family model has been manually assembled, the graph of family individuals and relationships will evolve in response to events.

2.1.1 Semantic Web Vision

The World Wide Web was invented in 1989 by Tim Berners-Lee, with the initial purpose to meet the demand for automatic information-sharing among members of scientific communities [7]. Its major breakthrough was the hyperlink, which allows linking of documents on a network of machines. In the "first generation" of Web implementation, machines and Web browsers retrieve and render the document content, and end users interpret the content. There is no effort on the part of

machines to understand the semantic meaning of the content.

The Semantic Web is an extension of the World Wide Web that aims to produce a semantic data structure which allows machines to access and share information, thus constituting a communication knowledge between machines, and automated discovery of new knowledge [14, 16, 32]. If that data is ever updated, some applications, such as those that refer to a large amount of data from many different sources, benefit enormously from this feature.

In order to accomplish its goal, the Semantic Web relies on mechanisms (i.e., markup languages) that enable the introduction, coordination, and sharing of the formal semantics of data, as well as an ability to reason and draw conclusions (i.e., inference) from semantic data obtained by following hyperlinks to definitions of problem domains (i.e., so-called ontologies).

2.1.2 Technical Infrastructure

Figure 2.1 illustrates the technical infrastructure that supports the Semantic Web vision, and the foundation upon which we hope to build our system-behavior models.

Each layer exploits and uses capabilities of the layers below. Briefly, the bottom layer is constructed of Uniform Resource Identifier (URI) and Unicode. URI and Unicode provide capability for identifying resources on the Web, linking documents, and representing multi-lingual languages. The extensible Markup Language (XML) provides the fundamental layer for representation and management of data

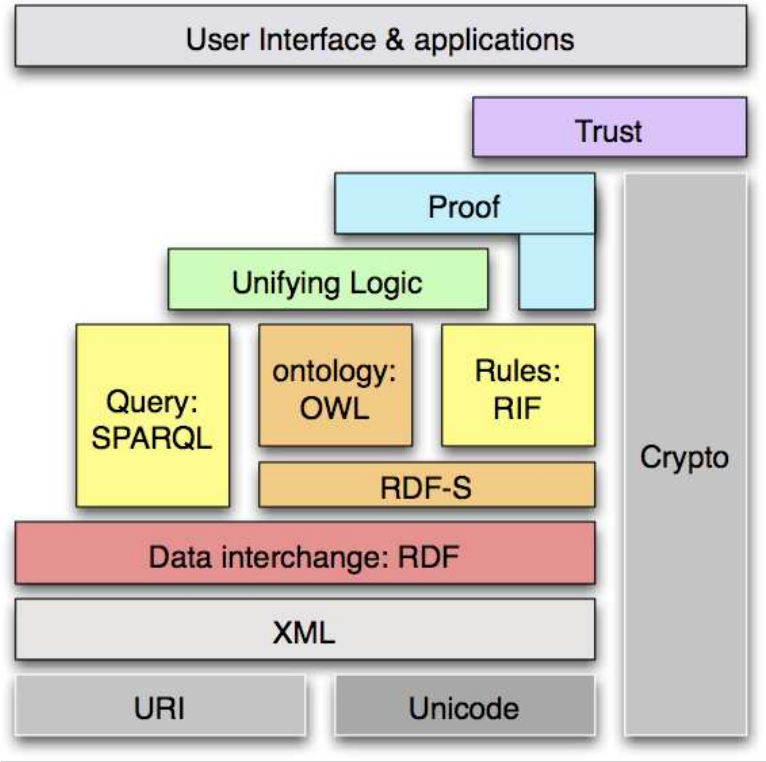

Figure 2.1: Technologies in Semantic Web Layer Cake [12].

on the Web. XML data is organized into tree hierarchies. As already noted, Semantic Web applications can gather information from a variety of sources, and in the context of our application, merge and organize these sources for decision making. Unfortunately, there is no easy way for tree structures to be merged. The resource description framework (RDF) solves this problem by allowing for the representation of graphs of data on the web. Graphs can always be merged. An RDF Schema (RDFS) provides the basic vocabulary for RDF. SPARQL is a RDF query language, it can be used to query any RDF-based data. The web ontology language (OWL) provides for semantic descriptions of the underlying data. Together, XML, RDF and OWL allow for the implementation of reasoning that can prove whether or not assertions are true or false.

2.2 Working with Semantic Web Technologies

2.2.1 Low-Level Technologies (URI and UNICODE)

At the bottom of the semantic web stack, unicode provides 16-bit support for multiple languages, and uniform resource identifiers (URI) provide a means for the unique identification of resources on the Web. Unicode enables the multi-language representation and handling of texts.

2.2.2 Extensible Markup Language (XML)

XML technology has two aspects. First, it is an open standard which describes how to declare and use simple tree-based data structures within a plain text file

(human readable format). XML is a meta-language (or set of rules) for defining domain- or industry-specific markup languages. Within the systems engineering community, for example, XML is being used in the implementation of AP233, a standard for exchange of systems engineering data among tools [26]. A second key benefit in representing data in XML is that we can filter, sort and re-purpose the data for different devices using the Extensible Stylesheet Language Transformation (XSLT) [35, 40].

2.2.3 Resource Description Framework (RDF)

While XML provides support for the portable encoding of data, it is limited to information that can be organized within hierarchical relationships. This can be a problematic situation for XML as a synthesized object may or may not fit into a hierarchical (tree) model. A graph, however, can, and thus we introduce the Resource Description Framework (RDF).

RDF is a graph-based assertional data model for describing the relationships between objects and classes (i.e., data and metadata) in a general but simple way, and for designating at least one understanding of a schema that is sharable and understandable. The graph-based nature of RDF means that it can resolve circular references, an inherent problem of the hierarchical structure of XML. An assertion is the smallest expression of useful information. RDF captures assertions made in simple sentences by connecting a subject to an object and a verb, as shown in Figure 2.2.

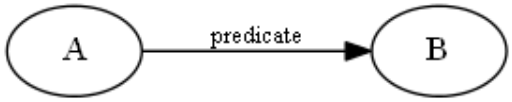

Figure 2.2: Example of RDF triple where node A is a subject, "predicate" is a verb, and node B is an object.

In practical terms, English statements are transformed into RDF triples consisting of a subject (this is the entity the statement is about), a predicate (this is the named attribute, or property, of the subject) and an object (the value of the named attribute). Subjects are denoted by a URI. Each property will have a specific meaning and may define its permitted values, the types of resources it can describe, and its relationship with other properties. Objects are denoted by a "string" or URI. The latter can be web resources such as requirements documents, other Web pages or, more generally, any resource that can be referenced using a URI (e.g., an application program or service program).

A set of related statements constitute an RDF graph. RDF graphs can be used to model a wide variety of relationships, including those among friends, location data, business data, and show information about a restaurant and a movie [32]. Figure 2.3 illustrates, for example, a graph model of relationships relevant to The Mona Lisa.

Limitations of RDF. Unfortunately, RDF is unable to capture vital knowledge attributes such as existence and cardinality or localized range and domain constraints as well as richer properties such as transitivity, inverse or symmetrical properties [18]. This makes it weaker to describe resources in sufficient detail and difficult in

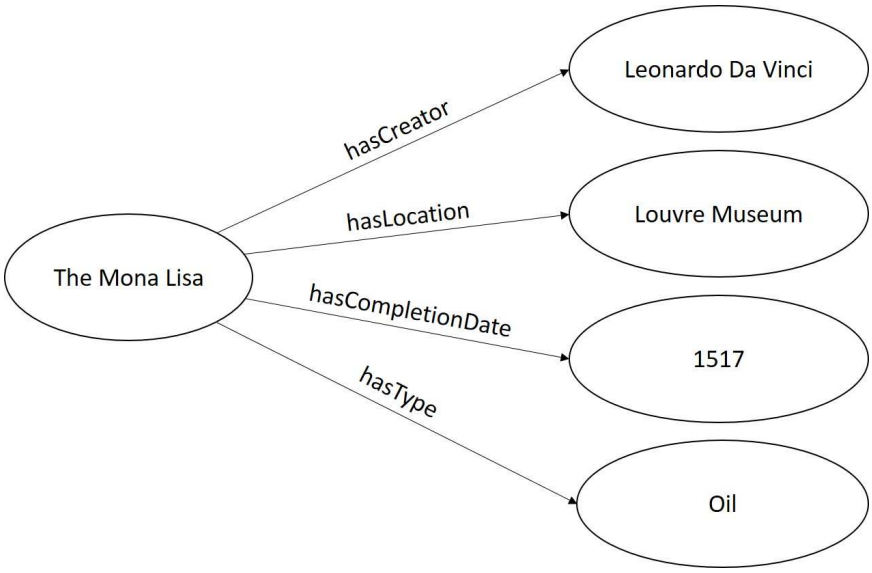

Figure 2.3: An RDF graph of relationships important to The Mona Lisa.

use to support reasoning. The Web Ontology Language (OWL) was developed to address the weaknesses of RDF [19].

2.2.4 The Web Ontology Language (OWL)

The Web Ontology Language (OWL) is a DL-based knowledge representation language for constructing ontologies. OWL is based on the basic features of RDF introduced above but it strengthens it by adding structure and vocabulary for describing properties and classes. They enable richer property definitions(e.g.: transitivity), class property restrictions(e.g.: allValuesFrom), and relationship between classes(e.g.: subClassOf). The additional capabilities allow ontological systems to use reasoning structures and infrastructure to infer new facts (triples) from existing ones with FOL as baseline mathematical, formal foundation. Below is an example of how the Mona Lisa example presented above can be translated into OWL. See Figure 2.4 and Figure 2.5.

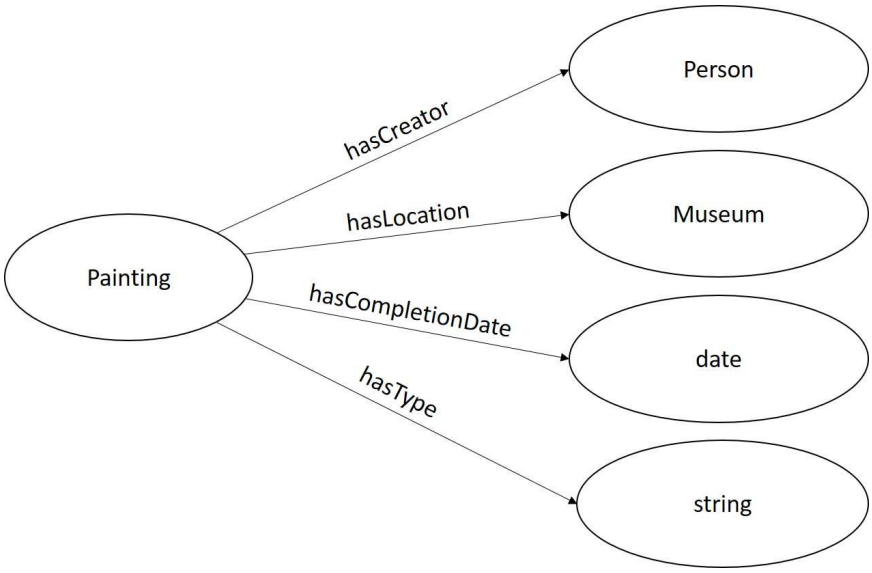

Figure 2.4: An OWL graph of relationships important to The Mona Lisa.

In the example, the class Painting, Person and Museum are defined. OWL can also define two types of properties: object properties and datatype properties. Object properties specify relationships between pairs of resources. Datatype properties, on the other hand, specify relation between a resource and a data type value; they are equivalent to the notion of attributes in some formalisms. In the example above, hasType and hasCompletionDate are defined as datatype properties, while hasCreator and hasLocation are defines as object properties. The rdfs:domain and rdfs:range properties are used to specify the domain and range of a property. The rdfs:domain of a property specifies that the subject of any statement using the property is a member of the class it specifies. Similarly, the rdfs:range of a property specifies that the object of any statement using the property is a member of the class or datatype it specifies.

The family of OWL encompasses three languages distinguished by their in-

```
// Define Classes ...

<owl:Class rdf:about="http://example.org/monaLisa#Painting">
</owl:Class>

<owl:Class rdf:about="http://example.org/monaLisa#Person">
</owl:Class>

<owl:Class rdf:about="http://example.org/monaLisa#Museum">
</owl:Class>

// Define Datatype Properties ...

<owl:DatatypeProperty rdf:about="http://example.org/monaLisa#hasType">
   <rdfs:domain rdf:resource="http://example.org/monaLisa#Painting"/>
   <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="http://example.org/monaLisa#hasCompletionDate">
   <rdfs:domain rdf:resource="http://example.org/monaLisa#Painting"/>
   <rdfs:range rdf:resource="&xsd;date"/>
</owl:DatatypeProperty>

// Define Object Properties ...

<owl:ObjectProperty rdf:about="http://example.org/monaLisa#hasCreator">
   <rdfs:domain rdf:resource="http://example.org/monaLisa#Painting"/>
   <rdfs:range rdf:resource="http://example.org/monaLisa#Person"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="http://example.org/monaLisa#hasLocation">
   <rdfs:domain rdf:resource="http://example.org/monaLisa#Painting"/>
   <rdfs:range rdf:resource="http://example.org/monaLisa#Museum"/>
</owl:ObjectProperty>
```

Figure 2.5: Formal definition of a “Famous Painting” in OWL.

creasing expressiveness. *OWL Lite* allows the expression of simple syntax and constraints but inferencing is more tractable using this version. *OWL DL* has a human-friendly syntax, inferencing is decidable and the language is computationally complete. *OWL Full* ensures full compatibility with RDF and RDFS languages however, the cost is that there is no guarantee in the validity of all computed statements[30].

2.3 Working with Jena and Jena Rules

Not all technologies on the semantic web are standardized. Some are emergent ones that are used mostly for horizontal and vertical integration of multiple layers of the stack. Generally speaking, there are Application Programming Interfaces (API) used to complete integration tasks.

2.3.1 Jena

Apache Jena [1] is an open source Java framework for building Semantic Web and linked data applications. Jena provides APIs (application programming interfaces) for developing code that handles RDF (resource description framework), RDFS, OWL (web ontology language) and SPARQL (support for query of RDF graphs). Jena uses a rule-based reasoning approach, which is the classic technique to logic-based reasoning where the knowledge-based system is developed by deduction, induction, abduction or choices from a starting set of data and rules. A unifying logic, such as the DL, is needed for horizontal integration of top layers of stacks and provide the rigorous, formal support needed by applications.

2.3.2 Jena Rules

The Jena inference subsystem is designed to allow a range of inference engines or reasoners to be plugged into Jena. Jena Rules is one such engine. Reasoners provide a means to derive additional RDF assertions which are entailed from some base RDF together with any optional ontology information and the axioms and rules associated with the reasoner. Jena Rules use facts and assertions described in OWL to infer additional facts from instance data and class descriptions. Such inferences result in structural transformations to the semantic graph model, as shown in Figure 2.7.

2.4 Simplified Modeling of Event-Driven Family Dynamics

This case study examines the work of Austin, Delgoshaei and Nguyen [3] from the perspective of basic ontology- and rule-based modeling of systems with Jena and Jena rules. Ontologies (Jena) and rules (Jena Rules) are defined for simplified behavior modeling of family dynamics. Once the family model has been manually assembled, the graph of family individuals and relationships will evolve in response to events.

2.4.1 Definition of the Family Ontology

Figure 2.6 shows a simplified family ontology, the relationship among classes and properties.

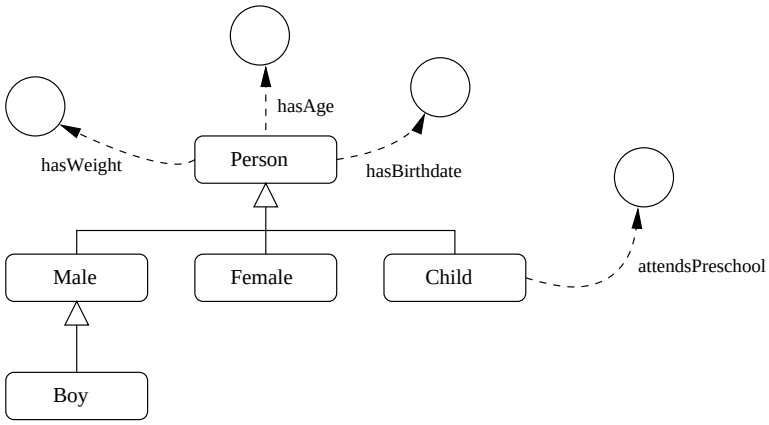

Figure 2.6: Relationship between classes and properties in a family ontology.

The ontology class Person has properties: hasAge, hasWeight, and hasBirthDate. They will be modeled as data types double, double and date, respectively. Male and Female are subclasses (specializations) of class Person. Boy is a specialization of Male. A Child is a Person who may (or may not) attend Preschool.

2.4.2 Adding Facts and Rules

To see how these ideas might work in practice, consider the following fact and small set of rules:

Fact 1: Sam is a boy born October 1, 2007.

The following rules can be declared:

Rule 1: For a given a birthdate and a current time, a built-in function `getAge()` computes a persons age.

Rule 2: A child is a person with age less than 18 .

Rule 3: Children who are age 5 attend preschool.

Figure 2.7 shows the evolution of a graph defining the properties of Sam as a function of time.

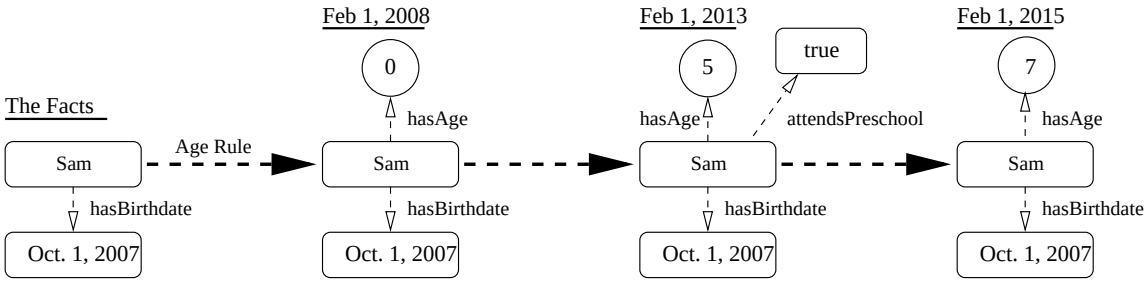

Figure 2.7: Evolution of ontology graph as a function of time.

Some of the data (e.g., Sams birthdate) remains constant over time. Other data (e.g., such as whether or not Sam attends preschool) is dynamic and is controlled by the family rules.

2.4.3 Definition and Organization of Ontology Classes

The abbreviated fragment of code below demonstrates the definition of the family ontology classes, their assembly into a hierarchy, and definition of data properties for the class `Person`.

```
// Define classes ...

person = model.createClass( ns + "Person");
male   = model.createClass( ns + "MalePerson");
boy    = model.createClass( ns + "Boy");

// Define relationships among classes ...

person.addSubClass (   male );
male.addSubClass (  boy );

// Create data properties for the class Person ...
```

```
hasAge = model.createDatatypeProperty( ns + "hasAge");
hasAge.setDomain(person);
hasAge.setRange( XSD.integer );

hasBirthDate = model.createDatatypeProperty( ns + "hasBirthDate");
hasBirthDate.setDomain(person);
hasBirthDate.setRange( XSD.date );
```

The data property `hasAge` is an integer. The data property `hasBirthDate` is a date. Notice that since `Boy` is a subclass of `MalePerson`, and `MalePerson` is a subclass of `Person`, boys automatically have the properties age and birthdate through class hierarchy inheritance.

2.4.4 Adding Individuals to the Family Model

The next step is to define family individuals, the data associated with each individual, and the relationship of one individual to other individuals in the family. The fragment of code below establishes a name space for the family ontology, creates a graph model for the storage of individuals and their data and object properties, and then creates an Individual model for `Sam` and a data property statement for his date of birth.

```
// Namespace for the family ontology ...

String ns = "http://austin.org/family#";

// Create ontology model (a graph) ...

OntModel model = ModelFactory.createOntologyModel();

// Add "Sam" to the family graph model ...

Individual sam = boy.createIndividual( ns + "Sam" );
model.add ( sam );
```

```
// Create statement: Sam's birthdate is 2007-10-01.

Literal bdate = model.createTypedLiteral( "2007-10-01", XSDDatatype.XSDdate );
Statement cbd = model.createStatement( sam, hasBirthDate, bdate );
model.add ( cbd );
```

Jena provides very powerful facilities for querying the graph model, subject to a wide range of search criteria.

2.4.5 Event-Driven Graph Transformations (Jena Rules)

Given the fact and three rules described above, graph transformations are enabled. `Sam` is a boy born October 1, 2007. Given a birthdate and a current time, a built-in function `getAge()` computes Sam's age. Further rules can be defined for when a person is also a child and when children attend Preschool. Figure 2.7 shows the evolution of a graph defining the properties of `Sam` as a function of time. The abbreviated fragment of code below is taken from the Jena Rules for the family ontology.

```
@prefix af:  <http://austin.org/family#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

// Rule 01: Propagate class hierarchy relationships ....

[ rdfs01: (?x rdfs:subClassOf ?y), notEqual(?x,?y) ->
          [ (?a rdf:type ?y) <- (?a rdf:type ?x)] ]

// Rule 02: Compute and store the age of a person ....

[ GetAge:    (?x rdf:type af:Person) (?x af:hasBirthDate ?y)
             getAge(?y,?z) -> (?x af:hasAge ?z) ]
[ UpdateAge: (?a rdf:type af:Person) (?a af:hasBirthDate ?b)
             (?a af:hasAge ?c) getAge(?b,?d) notEqual(?c, ?d) ->
             remove(2) (?a af:hasAge ?d) ]
```

The first rule propagates class hierarchy relationships. The second set of rules serves two purposes. First, given an individual's data of birth, the *GetAge* rule computes their age and inserts it into the semantic model via the `hasAge` data property. When a person has a birthday, the *UpdateAge* rule removes the old age from the graph and inserts the new age.

Chapter 3: **System Modeling and Software Architecture**

This chapter describes the system modeling assumptions and prototype software architecture for the generation and execution of semantic models and distributed system behaviors.

3.1 System Modeling Assumptions

This study takes an initial step toward creating the "city operating system" capability described in Figures 1.3 through 1.8. The simplifying assumptions are as follows:

1. Figure 1.11 shows a simplified view of the urban (semantic graph) domain models, their interfaces, and mechanisms for exchange of messages. The iterfaces serve two purposes. First, they listen for changes to the semantic graph within the domain – these changes could triggered, for example, through the execution of a domain-specific rule.

2. Each urban domain (semantic graph) has behavior that operates independently from the other domains. For the purposes of this study, however, we assume that all of the models execute under a single continuous thread of computation,

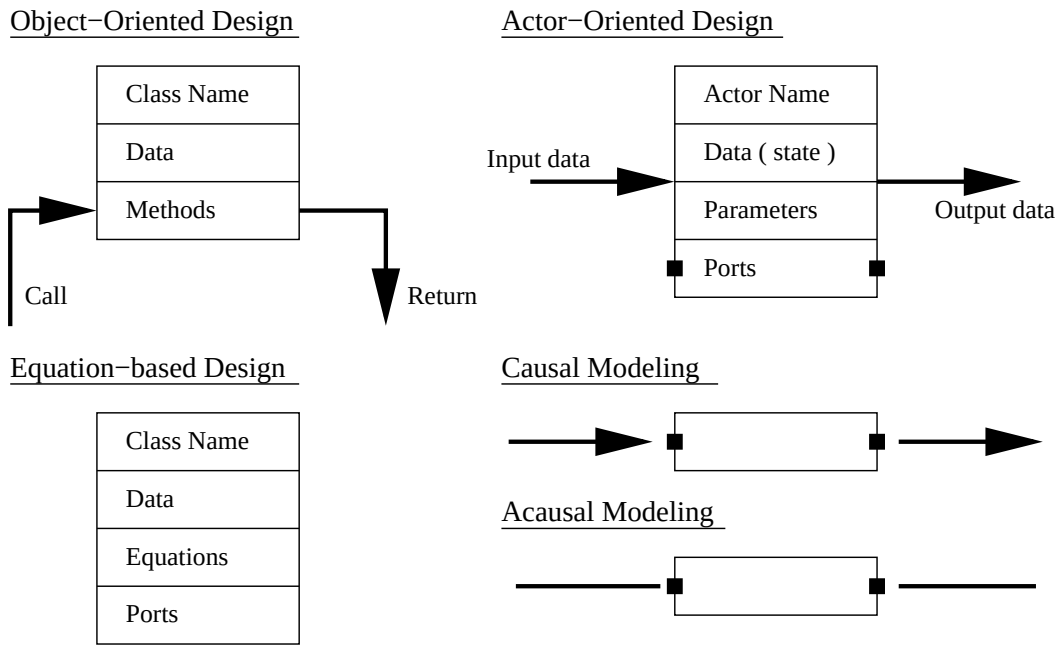

Figure 3.1: Five approaches to system/model development: (1) object-oriented, (2) actor-based, (3) equation-based, (4) causal modeling, and (5) acausal modeling [22].

with the only interaction among domains being exchange of messages. If we think of these messages as "flows of data" then the model of computation can be classified as being actor based (see Figure 3.1).

3. All of the participating domains operate under a single clock. Delays in communication between domains are ignored.

4. We assume that behavior models are deterministic. Uncertainties in behavior are ignored.

5. Support for fault-tolerant communication among domains is ignored. We do, however, send confirmation messages back to the sender.

3.2 Generation of Semantic Models

Figure 3.2 shows the pathway of development for generation of semantic models, consisting of ontologies, graphs of individuals (specic instances), and rules derived from engineering models.

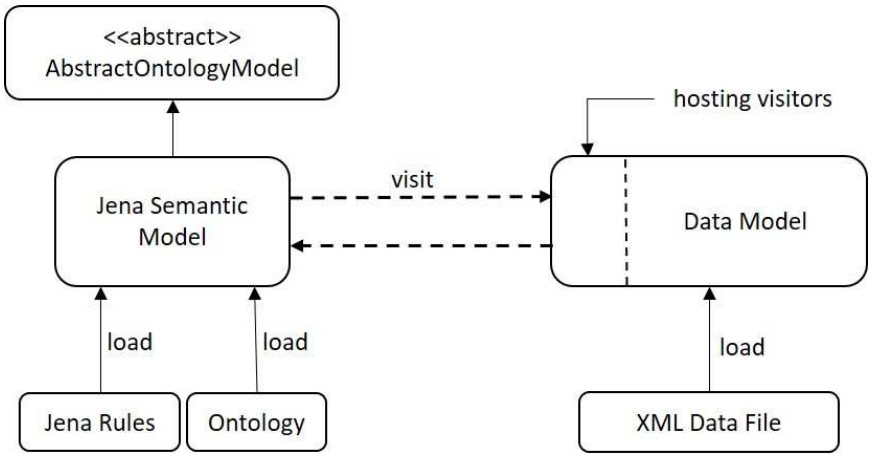

Figure 3.2: Pathway of development for generation of semantic models.

The process begins with the development of ontological descriptions of problem domains in OWL (the Web Ontology Language). Each ontology consists of a creating hierarchy of classes, and data and object properties. Next, we use the Jena Rules formalism to describe rules and represent domain-specific constraints. The data necessary to complete the model can be retrieved from an XML data file through a Data Model. The Data Model reads the XML data file and imports the data. Ontology, rules and data are all combined in the Jena Semantic Model. This semantic model creates an instance of the OWL ontology.

Note that the data in the data model may or may not pertain to the ontology instance in its entirety. Through the implementation of a visitor design pattern, the

data that does pertain to the ontology instance is transferred to the Jena Semantic Model, where the ontology and rules are applied to it.

In software engineering circles, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern is not a complete design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different applications. Design patterns can speed up the development process by providing tested, proven development paradigms. As a case in point, the visitor design pattern, is a way of separating an algorithm (i.e. system functionality) from an object structure on which it operates. This pattern should be used when distinct or unrelated operations are to be performed across a structure of objects. In the case of the semantic model described above, the system functionality to be separated from the semantic model is the retrieval of data pertaining to the ontology instance.

3.3 Distributed Behavior Modeling

Figure 3.3 shows the software architecture for distributed system behavior modeling for collections of graphs that have dynamic behavior defined by ontology classes, relationships among ontology classes, ontology and data properties, listeners, mediators and message passing mechanisms.

The abstract ontology model class contains concepts common to all ontologies (e.g., the ability to receive message input). Domain-specific ontologies are extensions of the abstract ontology classes. They add a name space and build the ontology

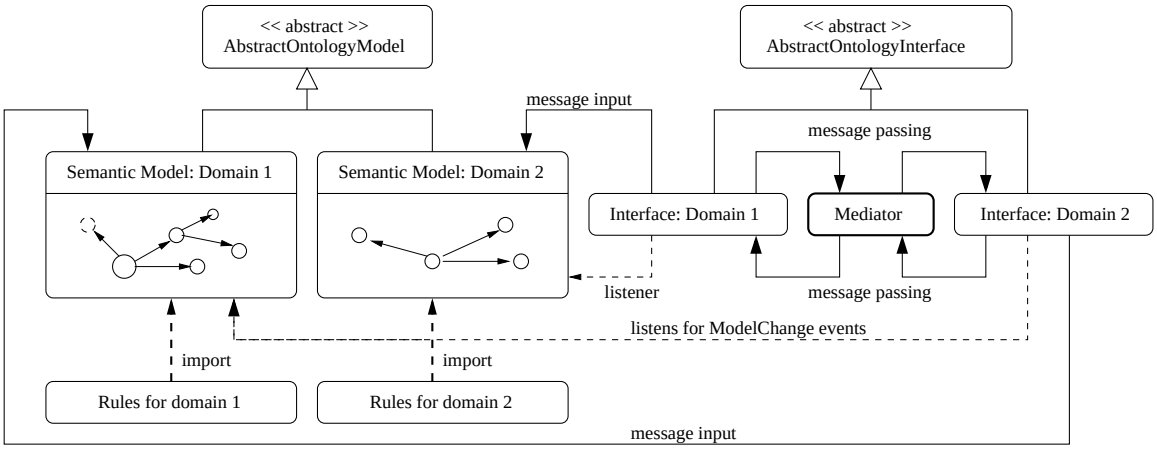

Figure 3.3: Software architecture for distributed behavior modeling in the family-school case study.

classes, relationships among classes, properties of classes for the domain. Instances (see Figure 1.6) are semantic objects in the domain.

Ontologies provide a framework for the representation of knowledge, but by themselves, cannot do much and really arent that interesting. This situation changes when domain-specific rules are imported into the model and graph transformations are enabled by formal reasoning and event-based input from external sources. Distributed behavior modeling involves multiple semantic models, multiple sets of rules, mechanisms of communication among semantic models, and data input, possibly from multiple sources. We provide this functionality in our distributed behavior model by loosely coupling each semantic model to a semantic interface. Each semantic interface listens for changes to the semantic domain graph and when required, forwards the essential details of the change to other domains (interfaces) that have registered interest in receiving notification of such changes. They also listen for incoming messages from external semantic models. Since changes to the graph structure are triggered by events (e.g., the addition of an individual; an update to a data

property value; a new association relationship among objects), a central challenge is design of the rules and ontology structure so that the interfaces will always be notified when exchanges of data and information need to occur. Individual messages are defined by their type (e.g., MessageType.miscellaneous), a message source and destination, and a reference to the value of the data being exchanged. The receiving interface will forward incoming messages to the semantic model, which, in turn, may trigger an update to the graph model. Since end-points of the basic message passing infrastructure are common to all semantic model interfaces, it makes sense to define it in an abstract ontology interface model.

3.4 Message Passing Mechanism

When the number of participating applications domains is very small, point-to-point channel communication between interfaces is practical. Otherwise, an efficient way of handling domain communication is by delegating the task of sending and receiving specific requests to a central object. In software engineering, a common pattern used to solve this problem is the Mediator Pattern.

As illustrated in Figures 1.4 and 1.11, the mediator pattern defines a object responsible for the overall communication of the system, which from here on out will be referred as the mediator object. The mediator has the role of a router, it centralizes the logic to send and receive messages. Components of the system send messages to the mediator rather than to the other components; likewise, they rely on the mediator to send change notifications to them [34]. The implementation

of this pattern greatly simplifies the other classes in the system; components are more generic since they no longer have to contain logic to manage communication with other components. Because other components remain generic, the mediator has to be application specific in order to encapsulate application-specific behavior. One can reuse all other classes for other applications, and only need to rewrite the mediator class for the new application.

Chapter 4: **Case Studies**

To illustrate the capabilities of our experimental software architecture, this chapter presents two case study problems. Case Study 1 describes behavior modeling of a multiplicity of families and school, defined by ontologies, rules, and exchange of information as messages. The decision making includes reasoning with time-driven events. In Case Study 2, decision making capability is extended to include reasoning with both space and time-driven events.

4.1 Case Study 1: Family-School System Dynamics

Figure 4.1 is an instantiation of the concepts introduced in Figure 3.3 and shows the software architecture for a family-school interaction. Figure 4.2 is the network setup for three families interacting with elementary, middle and high schools.

As every parent knows, the enrollment process involves the exchange of specific information, such as the name, birth date, home address and social security number of each child. Then, once the child is accepted the school system takes over. They figure out what grade level is appropriate for each child, what classroom the child will be in, the schedule of learning activities, and when school reports will be sent home.

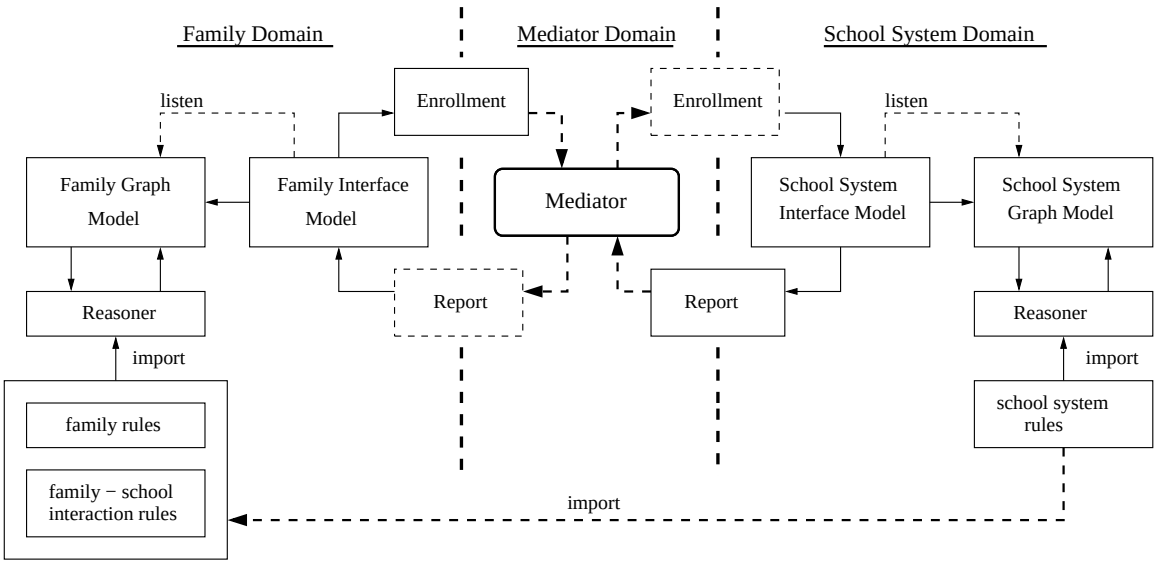

Figure 4.1: Framework for communication among multiple families and schools enabled by a mediator.

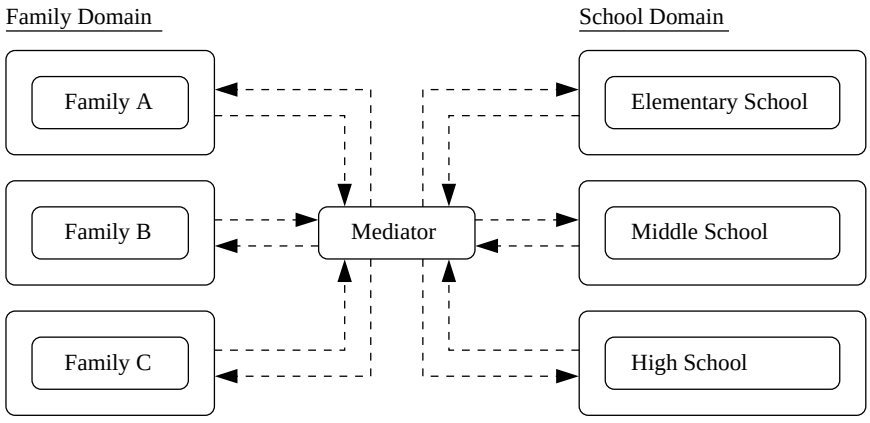

Figure 4.2: Framework for communication among multiple families and schools enabled by a mediator.

Communication among the family and school communities is handled by a mediator. Every component of the system (i.e., families and schools) register with the mediator as listeners. Once a family member reaches a certain age, the age rules associated with the family system will trigger a school enrollment form to be sent to the mediator in the form of a message, with source and destination properties. The mediator logic loops through all of its registered listeners to find a match with the message destination, and then destination listener is notified. Similarly, once the system calendar reaches a certain date, the reporting rules associated with the school system will trigger a school report to be sent to the mediator. The messaging design allows the school enrollment form to be received only by the school of interest, and not broadcasted to the entire school system. Likewise, this design allows the school reports to be sent only to the student's family. This mediator logic design is known as point-to-point channel, and it ensures that only one listener consumes any given message. The channel can have multiple listeners that consume multiple messages concurrently, but the design ensures that only one of them can successfully consume a particular message. Using this approach, listeners do not have to coordinate with each other; coordination could be complex, create a lot of communication overhead, and increase coupling between otherwise independent receivers.

4.1.1 Family and School Data Models

In this setup, the information to be exchanged between ontologies is contained in XML datafiles. Complete descriptions of the XML data for the family and school

system models are located in Appendix B. The abbreviated details are found below.

Family Data Model. The family data model defines the attributes of a family (e.g, name, address) and the persons who are members of the family.

```
<?xml version="1.0" encoding="UTF-8"?>
<FamilyModel author="Maria Coelho" date="2017" source="UMD">

<Family>
    <attribute text="FamilyName" value="Austin"/>
    <attribute text="Address" value="6242 Heather Glen Way, Clarksville, MD 21029"/>
    <Person>
        <attribute text="Type" value="Male"/>
        <attribute text="FirstName" value="Mark"/>
        <attribute text="MiddleName" value="William"/>
        <attribute text="LastName" value="Austin"/>
        <attribute text="BirthDate" value="1704-06-10"/>
        <attribute text="Weight" value="170.0"/>
        <attribute text="Citizenship" value="New Zealand"/>
        <attribute text="SocialSecurity" value="111"/>
    </Person>
    <Person>
        ... description of Christopher Austin ....
    </Person>
    <Person>
        ... description of Nina Austin ....
    </Person>
    <Person>
        ... description of Erin Austin ....
    </Person>
</Family>

<Family>
    <attribute text="FamilyName" value="Jones"/>
    <attribute text="Address" value="5807 Laurel Leaves Ln, Clarksville, MD 21029"/>
    <Person>
        ... description of Robert Jones ....
    </Person>
    <Person>
        ... description of Timothy Jones ....
    </Person>
    <Person>
        ... description of Samantha Jones ....
    </Person>
</Family>
</FamilyModel>
```

In the case of the family ontology, the XML datafile describes information about the individual families and their corresponding family members. The information is stored as key/value pairs. The key (e.g. "first name", "citizenship", etc.) identifies, and is used to retrieve, the value (e.g. "Mark", "New Zealand", etc.). In the same fashion, the school system XML datafile describes information about the individual schools.

School System Data Model. The school system data model defines the grade levels that will be taught at each school and the interval of time when reports will be sent home.

```
<?xml version="1.0" encoding="UTF-8"?>
<SchoolSystemModel author="Maria Coelho" date="2017" source="UMD">
   <School>
       <attribute text="Type" value="High School"/>
       <attribute text="Name" value="River Hill High School"/>
       <attribute text="Grade" value="Grade09"/>
       <attribute text="Grade" value="Grade10"/>
       <attribute text="Grade" value="Grade11"/>
       <attribute text="Grade" value="Grade12"/>
       <attribute text="Report Period Start Time" value="2016-09-01T00:00:00"/>
       <attribute text="Report Period End Time" value="2020-10-20T00:00:00"/>
   </School>
   <School>
       ... description of Clarksville Middle School ...
   </School>
   <School>
       ... description of Pointers Run Elementary School ...
   </School>
</SchoolSystemModel>
```

Instantiating the Family and School Data Models. As they are, XML data files are basically text. However, value is added to the information once it can be extracted from the XML file and instantiated (i.e., text file is converted into

class instances). Because the XML datafiles store information in a structured way, information retrieval is facilitated. In order to perform the data retrieval, we used a Java API called JAXB. JAXB stands for Java architecture for XML binding. It is used to convert XML to Java object through a process called Marshalling, and Java object to XML through a process called Unmarshelling. For our application, we use Unmarshelling to read the XML files, and create a Data Model as shown in Figure 3.2. Then, we create instances of the ontology classes, laden with the data from XML files.

4.1.2 Family and School Ontology Models

Ontologies are defined by classes, data and object properties, and the relationships among them. Our application employs the Web Ontology Language (OWL) to define ontologies. Complete descriptions of the OWL files for the family and school system models are located in Appendix C. The abbreviated details are shown Figure 4.3 and Figure 4.4.

Family Ontology. Figure 4.3 shows the relationship between classes in a family ontology. Male, Female, Child and Student are subclasses of class Person. The class Boy is a subclass of class Male. The class Person has properties that get inherited by all subclasses such as hasAge, hasWeight, hasBirthdate, hasFamilyName, hasFirstName, hasSocialSecurityNo, hasCitizenship. The class Student has properties associated with school enrollment, such as attendsPreschool, attendsSchool, attendsElementarySchool, attendsMiddleSchool, attendsHighSchool, and hasReportFrom.

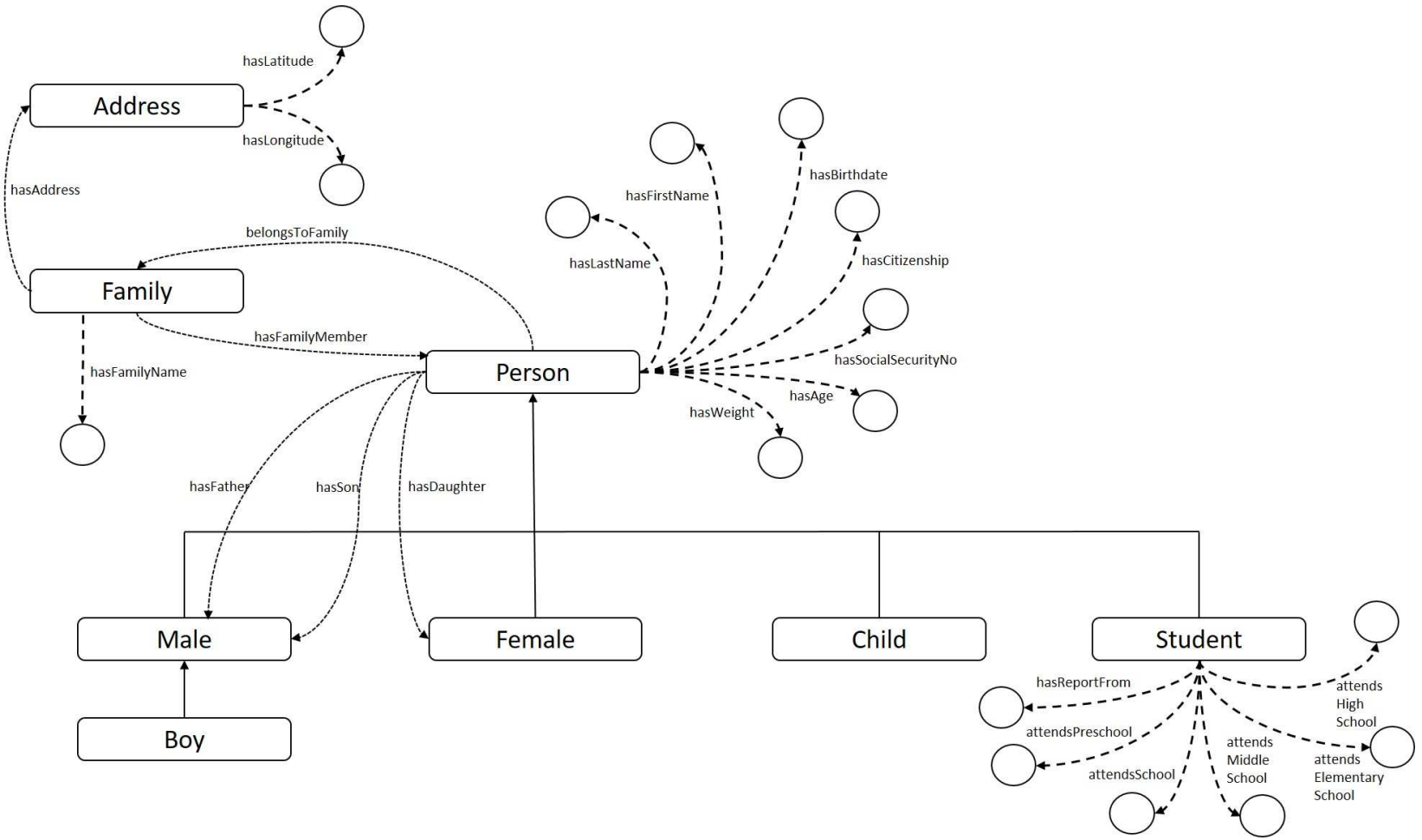

Figure 4.3: Family ontology diagram with classes, properties, and relationships among them.

The class family has property hasFamilyName, and the class Address has properties hasLatitude and hasLongitude. Other properties such as hasFamilyMember, belongsToFamily, hasFather, hasSon, hasDaughter, and hasAddress define relationships that hold between objects.

School System Ontology. In the same fashion, an ontology can be constructed for the school system. Figure4.4 shows the relationship between classes in a school ontology. Elementary School, Middle School and High School are subclasses of School. Grades 1 through 12 are subclasses of Grade. A school has properties that get inherited by all school subclasses such as hasName. A grade also has properties that get inherited by all grade subclasses such as hasEnrollment. A student has properties similar to the ones defined in the classes Person and Student in the family ontology such as hasFirstName, hasFamilyName, hasBirthDate, hasAge, hasSocialSecurityNo, attendsElemntarySchool, attendsMiddleSchool, attendsHighSchool, and hasReport. The class Address also follows the same pattern of the family ontology, with properties hasLatitude and hasLongitude. The classes Calendar and Event are included in this ontology to provide temporal behavior modeling capabilities. The class Event has properties hasStartTime and hasEndTime. Other properties such as hasGrade, hasStudent, isInGrade, hasStudentAddress, hasSchoolAddress and hasEvent define relationships that can hold between objects.

Instantiating the Family and School Ontology Models. To instantiate these ontologies with information retrieved from the XML datafiles, a visitor object is used to visit the Data Model, retrieve the data, and create instances of ontology

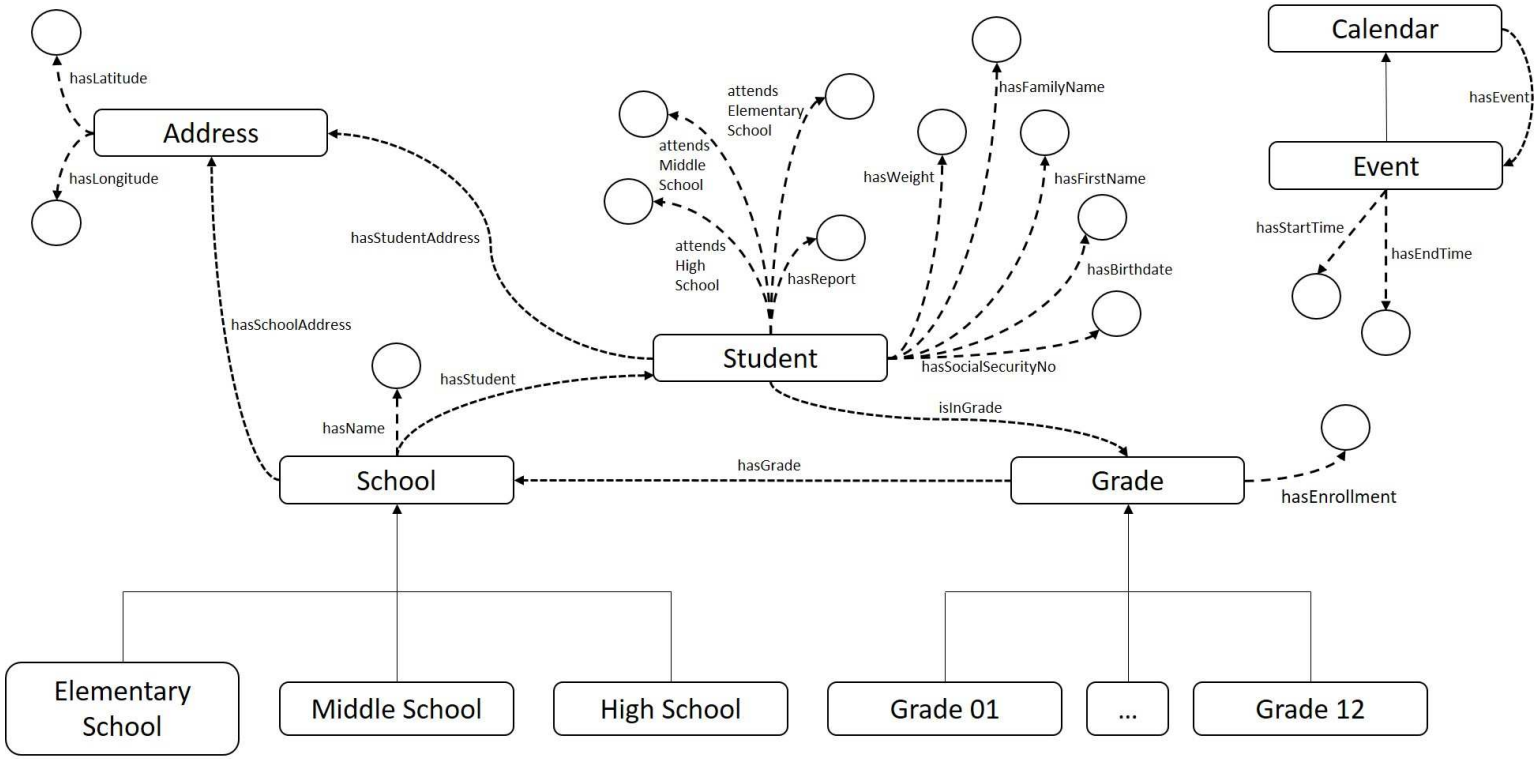

Figure 4.4: School ontology diagram with classes, properties, and relationships among them.

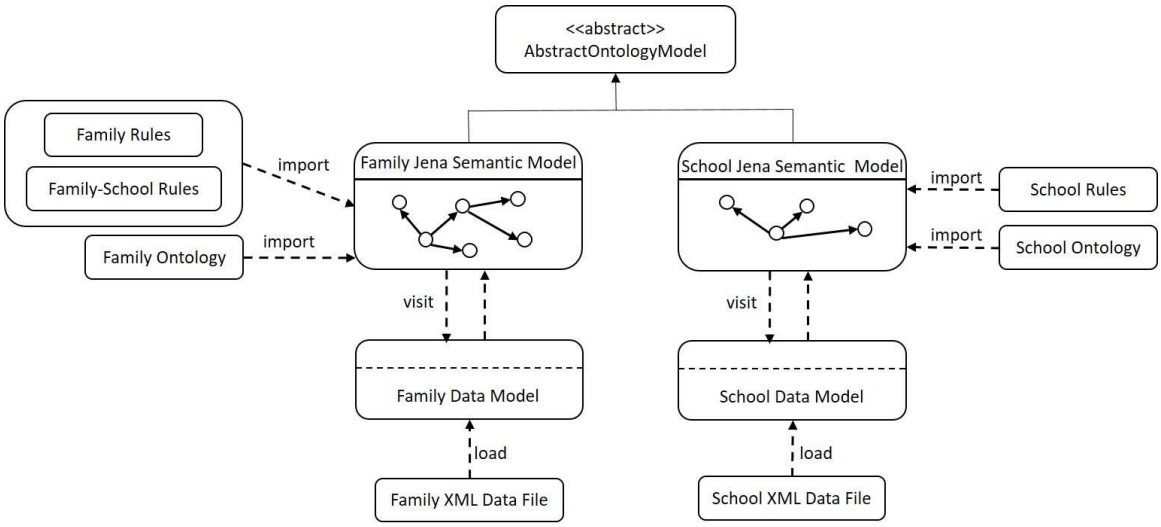

Figure 4.5: Generation of family and school semantic models.

classes with the data, as described in Figure 4.5. Because the data contained in the Data Models pertains to multiple instances of the ontologies (i.e. Family Data Model contains information about multiple families), the Data Model design assures visiting objects can only retrieve data pertaining to their corresponding ontology instance through a "password" mechanism. That way privacy of the different instances is preserved. The data retrieved by the visitor object is then used to create instances of the ontology classes in the Jena Semantic Models.

4.1.3 Family and School Jena Rules

Ontologies by themselves provide a framework for the representation of knowledge, but otherwise cannot model the dynamic evolution of objects, properties and relationships. Consider the family ontology, some of the data remains constant over time (e.g., birthdates), while other data is dynamic (e.g., attending preschool). However, when coupled with a set of domain-specific rules, ontological representa-

tions enable graph transformations by formal reasoning and event-based input from external sources. In our application, we use Jena Rules to define domain-specific rules. Complete descriptions of the rules for the family and school system models can be found in the appendices – abbreviated details are presented in Figure 4.6.

Family Rules. Figure 4.6 contains an abbreviated list of Jena rules for identifying relationships and properties within a family semantic model. The combination of ontologies and ontology rules is extremely powerful in scenarios where ontology graphs are dynamic. For example, the boy Christopher was born December 10,2007. Given a birthdate and the current year, a built-in function getAge() compute Christophers age. An age rule defined using Jena Rules determines whether or not a person is also a child. Therefore, the behavior modeling for the family system is defined by the set of rules governing graph transformations. Graph transformation can occur due to input (e.g. family graph changes because a new child is born) or time (e.g. family graph changes because a specific member has not the age of a child anymore).

School Rules. Figure 4.7 contains an abbreviated list of Jena rules for transformation of the School Semantic Model. Rules are provides for attendance, progression through the grades and timing of school reports.

The combination of ontologies and ontology rules is also extremely powerful in scenarios where ontology graphs are event dependent. For example, consider the boy Christopher again, now identified as a student from the school system perspective. A built-in function getToday() computes the current date. A rule defined using Jena Rules determines whether or not a student has a report by comparing the

```
@prefix af:  <http://austin.org/family#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

// Rule 01: Propagate class hierarchy relationships ....

[ rdfs01: (?x rdfs:subClassOf ?y), notEqual(?x,?y),
          (?a rdf:type ?x) -> (?a rdf:type ?y)]

// Rule 02: Family rules ....

[ Family: (?x rdf:type af:Family) (?x af:hasFamilyMember ?y) ->
          (?y af:belongsToFamily ?x) ]

// Rule 02: Identify a person who is also a child ...

[ Child: (?x rdf:type af:Person) (?x af:hasAge ?y)
         lessThan(?y, 18) -> (?x rdf:type af:Child) ]

[ UpdateChild: (?x rdf:type af:Child) (?x af:hasBirthDate ?y)
               getAge(?y,?b) ge(?b, 18) -> remove(0) ]

// Rule 03: Identify a person who is also a student ...

[ Student: (?x rdf:type af:Person) (?x af:hasAge ?y)
      greaterThan(?y, 4) lessThan(?y, 18) -> (?x rdf:type af:Student) ]

[ UpdateStudent: (?x rdf:type af:Student) (?x af:hasBirthDate ?y)
      getAge(?y,?b) ge(?b, 18) -> remove(0) ]

// Rule 04: Compute and store the age of a person ....

[ GetAge: (?x rdf:type af:Person) (?x af:hasBirthDate ?y)
          getAge(?y,?z) -> (?x af:hasAge ?z) ]

[ UpdateAge: (?a rdf:type af:Person) (?a af:hasBirthDate ?b) (?a af:hasAge ?c)
          getAge(?b,?d) notEqual(?c, ?d) -> remove(2) (?a af:hasAge ?d) ]

// Rule 05: Set father-son and father-daughter relationships ...

[ SetFather01: (?f rdf:type af:Male) (?f af:hasSon ?s)-> (?s af:hasFather ?f)]

[ SetFather02: (?f rdf:type af:Male) (?f af:hasDaughter ?s)-> (?s af:hasFather ?f)]
```

Figure 4.6: Abbreviated list of Jena rules for transformation of the Family Semantic Model.

```
@prefix af:  <http://austin.org/school#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

// Rule 01: Propagate class hierarchy relationships ....

[ rdfs01: (?x rdfs:subClassOf ?y), notEqual(?x,?y),
          (?a rdf:type ?x) -> (?a rdf:type ?y)]

// Rules 02: Elementary school rules ...

[ EnterElementarySchool: (?x rdf:type af:Student) (?y rdf:type af:ElementarySchool)
     (?x af:hasBirthDate ?a) getAge(?a,?b) ge(?b, 6) le(?b, 10) ->
     (?x af:attendsElementarySchool af:True) (?y af:hasStudent ?x)]

[ LeaveElementarySchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
     (?x af:attendsElementarySchool af:True) (?y af:hasStudent ?x)
     getAge(?a,?b) ge(?b, 10) -> remove(2) ]

[ GradeOne:   (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
              getAge(?a,?b) equal(?b, 6) -> (?x af:isInGrade af:Grade01) ]
[ GradeTwo:   (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
              getAge(?a,?b) equal(?b, 7) -> (?x af:isInGrade af:Grade02) ]

... Rules for Grades 3 through 5 removed ...

// Rules 03: Middle school rules ...

... Middle school rules removed ...

// Rules 04: High school rules ...

[ EnterHighSchool: (?x rdf:type af:Student) (?y rdf:type af:HighSchool)
     (?x af:hasBirthDate ?a) getAge(?a,?b) ge(?b, 14) le(?b, 17) ->
     (?x af:attendsHighSchool af:True) (?y af:hasStudent ?x) ]

[ LeaveHighSchool: (?x rdf:type af:Student) (?y rdf:type af:HighSchool)
     (?x af:hasBirthDate ?a) (?x af:attendsHighSchool af:True)
     (?y af:hasStudent ?x) getAge(?a,?b) ge(?b, 17) -> remove(2) ]

... Rules for Grades 9 through 12 removed ...

// Rules 05: If today is report period, send school report ....

[ GenerateReport:  (?x rdf:type af:Event) (?y rdf:type af:Student)
     (?z rdf:type af:School)  (?z af:hasStudent ?y) (?x af:hasStartTime ?t1)
     (?x af:hasEndTime ?t2) getToday(?t3) lessThan(?t3,?t2)
     greaterThan(?t3,?t1) -> (?y af:hasReport af:True)  ]
```

Figure 4.7: Abbreviated list of Jena rules for transformation of the School Semantic Model.

output of the built-in function getToday() to the start and end dates of reporting period events. Therefore, similar to the family system, the behavior modeling for the school system is defined by the set of rules governing graph transformations. Graph transformation can occur due to input (e.g., school graph changes because a student is enrolled) or time (e.g., school graph changes because a reporting period has started).

Rules for Family-School System Interaction. So far, the family and school rule systems have been completely decoupled and one might think that they operate independently. In reality, a small set of rules that govern family behavior are actually defined by the school system. When a child is old enough to attend preschool is one example.

Figure 4.8 contains an abbreviated list of rules for family-school system interactions. Three set of rules are needed. First a set of rules pertaining just to the family model, a set of rules pertaining just to the school model, and a set of rules pertaining to both the school and family models, which from now on will be referenced as family-school rules. This family-school interface allows the school system to distribute relevant rules to the family system. For example, consider the situation where Christopher is now old enough to attend regular school. The family-school set of rules will inform Christophers family that he is now old enough to attend regular school by triggering a change to the family graph. This change, in turn, will trigger the school enrollment process for Christopher to start.

```
// ================================================================
// School-family interaction rules ...
// ================================================================

@prefix af:  <http://austin.org/family#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

// Rules 01: Children of age 4 and 5 attend preschool ...

[ EnterPreSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a) getAge(?a,?b)
       ge(?b, 4) le(?b, 5) -> (?x af:attendsPreSchool af:True) ]

[ LeavePreSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
       (?x af:attendsPreSchool af:True) getAge(?a,?b) ge(?b, 6) -> remove(2) ]

// Rules 02: Children aged 6 through 10 attend elementary school ....

[ EnterSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
       getAge(?a,?b) ge(?b, 6) le(?b, 10) -> (?x af:attendsElementarySchool af:True) ]

[ LeaveSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
       (?x af:attendsElementarySchool af:True) getAge(?a,?b) ge(?b, 11) -> remove(2) ]

// Rules 03: Children aged 11 through 13 attend middle school ....

... Rules for attending Middle school removed ...

// Rules 04: Children aged 14 through 17 attend high school ....

[ EnterSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
       getAge(?a,?b) ge(?b, 14) le(?b, 17) -> (?x af:attendsHighSchool af:True) ]

[ LeaveSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
       (?x af:attendsHighSchool af:True) getAge(?a,?b) ge(?b, 18) -> remove(2) ]

// Rules 05: Children aged 6 through 18 attend regular school ....

[ EnterSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a) getAge(?a,?b)
       ge(?b, 6) le(?b, 17) -> (?x af:attendsSchool af:True) ]

[ LeaveSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
       (?x af:attendsSchool af:True) getAge(?a,?b) ge(?b, 18) -> remove(2) ]
```

Figure 4.8: Abbreviated list of Jena rules for family-school system interactions.

4.1.4 Assembly of the Family-School Simulation Model

The step-by-step procedure for assembly of the family-school simulation model is as follows:

Step 01. Create empty semantic graph models, then load ontologies. The fragment of code creates semantic (graph-based) models for two families and three schools:

```
JenaFamilySemanticModel austin = new JenaFamilySemanticModel();
JenaFamilySemanticModel  jones = new JenaFamilySemanticModel();

JenaSchoolSystemSemanticModel hs = new JenaSchoolSystemSemanticModel();
JenaSchoolSystemSemanticModel ms = new JenaSchoolSystemSemanticModel();
JenaSchoolSystemSemanticModel es = new JenaSchoolSystemSemanticModel();
```

Family semantic models are handled by the class JenaFamilySemanticModel. School system semantic models are handled by the class JenaSchoolSystemSemanticModel. Both JenaFamilySemanticModel and JenaSchoolSystemSemanticModel are extensions AbstractSemanticModel. For simplicity of implementation, code for loading the family and school system ontologies (e.g., umd-family.owl and umd-school-system.owl) are embedded within the class constructors.

Step 02. Connect family and school system models. Each component of the system (i.e., families and schools) is paired with an interface for communication and information exchange with other components. Every component also registers with the mediator as message listeners, as described in Section 4.1. The fragment

of code:

```
// Retrieve family and school system interfaces ..

JenaFamilySemanticModelInterface austinfmi = austin.getModelInterface();
austinfmi.setType("Austin");

JenaFamilySemanticModelInterface  jonesfmi = jones.getModelInterface();
jonesfmi.setType("Jones");

JenaSchoolSystemSemanticModelInterface hsi = highschool.getModelInterface();
hsi.setType("River Hill High School");

JenaSchoolSystemSemanticModelInterface msi = middleschool.getModelInterface();
msi.setType("Clarksville Middle School");

JenaSchoolSystemSemanticModelInterface esi = elementaryschool.getModelInterface();
esi.setType("Pointers Run Elementary School");

// Add message listeners to family interfaces ...

austinfmi.addMessageListener ( (MessageListener) hsi );
austinfmi.addMessageListener ( (MessageListener) msi );
austinfmi.addMessageListener ( (MessageListener) esi );

jonesfmi.addMessageListener ( (MessageListener) hsi );
jonesfmi.addMessageListener ( (MessageListener) msi );
jonesfmi.addMessageListener ( (MessageListener) esi );

// Add message listeners to school interfaces ...

hsi.addMessageListener ( (MessageListener) austinfmi );
hsi.addMessageListener ( (MessageListener) jonesfmi );

msi.addMessageListener ( (MessageListener) austinfmi );
msi.addMessageListener ( (MessageListener) jonesfmi );

esi.addMessageListener ( (MessageListener) austinfmi );
esi.addMessageListener ( (MessageListener) jonesfmi );
```

retrieves references to the family and school system interfaces, and then systematically adds message listeners to the family and school system interfaces. The family interfaces listen for incoming messges from the schools. And the school interfaces listen for incoming messages from the families. In the background, the bi-directional

routing of messages (from source to destination) is handled by the message mediator.

Step 03. Create visitor object models. The result of Step 02 is a network of connected semantic graphs that are populated with ontologies, but do not contain individuals. The data for the family and school systems individuals is contained in XML files (for details, see Appendix B). We populate the semantic models with data on individual families and schools by creating visitor objects models that will be given permission to visit and extract information from the family and school system data models. This is the visitor software design pattern in action.

The fragment of code:

```
FamilyDataModelJenaVisitor austin_visitor = new FamilyDataModelJenaVisitor();
austin_visitor.setPassword("Austin");
austin_visitor.addSemanticModel( austin );

FamilyDataModelJenaVisitor jones_visitor = new FamilyDataModelJenaVisitor();
jones_visitor.setPassword("Jones");
jones_visitor.addSemanticModel( jones );

SchoolSystemDataModelJenaVisitor hsv = new SchoolSystemDataModelJenaVisitor();
hsv.setPassword("River Hill High School");
hsv.addSemanticModel( hs );

SchoolSystemDataModelJenaVisitor msv = new SchoolSystemDataModelJenaVisitor();
msv.setPassword("Clarksville Middle School");
msv.addSemanticModel( me );

SchoolSystemDataModelJenaVisitor esv = new SchoolSystemDataModelJenaVisitor();
esv.setPassword("Pointers Run Elementary School");
esv.addSemanticModel( es );
```

creates the visitor objects for the family and schools, sets the appropriate visitor passwords, and adds to each visitor references to the appropriate semantic graph. Notice how each visitor uses a password mechanism to retrieve only the data that

is of interest to their semantic model.

Step 04. Get data from XML files. This step creates data models for the family and school ontologies, and populates them with data imported from XML datafiles. The fragment of code:

```
FamilyDataModel fdm = new FamilyDataModel();
fdm.getData( "data/FamilyModel.xml" );

SchoolSystemDataModel ssdm = new SchoolSystemDataModel();
ssdm.getData( "data/SchoolSystemModel.xml" );
```

creates FamilyDataModel and SchoolSystemDataModel objects to store the data that will be imported from the datafiles FamilyModel.xml and SchoolSystemModel.xml, respectively.

Step 05. Populate semantic models with individuals. The family and school semantic graphs are populated with individuals by visiting the family and school system data models, respectively. In the fragment of code:

```
fdm.accept ( austin_visitor ); //Semantic model visits the data model ...
fdm.accept ( jones_visitor );

ssdm.accept ( hsv );
ssdm.accept ( msv );
ssdm.accept ( esv );
```

the family data model accepts the Austin and Jones semantic models as visitors. Similarly, the school system data model accepts semantic graph models of schools

as visitors. The password information provided at Step 03 ensures that semantic models for the individual families and schools are correctly matched with their data models.

Step 06. Add rules. Then execute. Finally, we import rules into each of the semantic models, and trigger transformations to the graph models by executing them. The fragment of code:

```
// Add rules to the school models ...

hs.addRules ( "src/demo/rules/schoolRules.rules" );
ms.addRules ( "src/demo/rules/schoolRules.rules" );
es.addRules ( "src/demo/rules/schoolRules.rules" );

// Transform school system semantic graph models ...

hs.executeRules();
ms.executeRules();
es.executeRules();

// Add family and family-school interaction rules to family semantic models ...

austin.addRules ( "src/demo/rules/familyRules.rules",
                  "src/demo/rules/familyschoolRules.rules" );
jones.addRules  ( "src/demo/rules/familyRules.rules",
                  "src/demo/rules/familyschoolRules.rules" );

// Transform family semantic graph models ...

austin.executeRules();
jones.executeRules();
```

adds the contents of schoolRules.rules to the school semantic graphs, and familyRules.rules and familyschoolRules.rules to the family semantic graph models. The familyRules.rules are sets of rules that trigger transformations to the family semantic model. The familyschoolRules.rules are sets of rules that a family semantic model

needs to know in order to interact with the school system. Execution of the rules within each semantic model occurs with the method calls `jones.executeRules()` and so forth.

At the conclusion of Step 06, interfaces are listening for changes to the graphs of semantic models to which they are attached to. Once a change is identified, the interface will communicate that information to other interested semantic models in the form of a message. The message will have a type, source and destination. The mediator will match the message destination to the corresponding interface among the registered listening interfaces, and forward the message. Once the receiving interface receives the message, it triggers changes in the graph of the semantic model it is attached to, and the process may start again.

4.1.5 Simulation of Family-School Interactions

In this case study, interactions between the family and school systems occur in response to time-driven events. For example, the family-school interaction rules define a range of ages within which child should be enrolled in elementary school. When a child's age falls within the acceptable range, the boolean property "attendsElementarySchool" will be set to "True". The family's semantic model interface will identify the corresponding update to the semantic graph, and in response, send an enrollment request to the elementary school in the form of a message, containing relevant information such the childs first name, last name, social security number and date of birth.

Task 01. Enrolling a Child in School. The following fragment of output shows the essential details of a school enrollment request:

```
*** Entering JenaFamilySemanticModelInterface.addStatement(Statement s) ...
*** ========================================================================
*** Subject:   = http://www.ontologies.org/family#Christopher
*** Predicate: = http://austin.org/family#attendsElementarySchool
*** Object:    = http://austin.org/family#True
*** Subject local name:   s = Christopher
*** Predicate local Name: s = attendsElementarySchool
*** Object resource:      s = True
*** Object value = null
*** Kid attends school: Christopher
message handlerMessage:
Type      = NEW_ELEMENTARY_SCHOOL_STUDENT
Source    = Austin
Destination = Pointers Run Elementary School
Subject   = --- Here's a new kid for your elementary school ...
Body      = School Enrollment Form ...
========================================
First Name = Christopher
Last Name  = Austin
Date of Birth = Fri Oct 12 00:00:00 EDT 2007
Social Security No = 678
========================================
```

The mediator will match the message destination, Pointers Run Elementary School, with the elementary school's semantic model interface and forward the message. The school semantic model interface will identify the message type (i.e. new elementary school student), and trigger changes to the semantic model graph by adding Christopher to the list of students.

Task 02. School Enrollment Acceptance. The elementary school's semantic model interface is listening for changes to the semantic model graph. Once Christopher is enrolled, the interface sends a message back to the Austin family to confirm

the enrollment. Here are the essential details:

```
*** Entering JenaSchoolSystemSemanticModelInterface.addStatement(Statement s) ...
*** ================================================================
*** Subject:   = http://www.ontologies.org/school#Pointers Run Elementary School
*** Predicate: = http://austin.org/school#hasStudent
*** Object:    = http://www.ontologies.org/school#Christopher
*** Subject local name:   s = Pointers Run Elementary School
*** Predicate local Name: s = hasStudent
*** Object resource:      s = Christopher
*** Object value = null
*** New student enrolled : Christopher
message handlerMessage:
Type   = NEW_STUDENT_ENROLLED
Source = Pointers Run Elementary School
Destination = Austin
Subject = --- To: Austin family ...
--- Message: Christopher is now enrolled with Pointers Run Elementary School...
Body    = Enrollment Confirmation Form ...
========================================
First Name = Christopher
Last Name  = Austin
School Name = Pointers Run Elementary School
========================================
```

Again, the mediator will match the message destination (in this case, Austin family), with family's semantic model interface, and forward the message. The interface for the family semantic model will identify the incoming message type (i.e. new student enrolled). No further action is needed from the family side, and no changes to the semantic model graph are triggered.

Task 03. Sending the School Reports Home. Student reports are sent home during the school reporting period. This is a time-driven event. The school rules establish that the school reporting period has started, and so the boolean property "hasReport" becomes "True". The elementary school's semantic model interface

will identify such a change and send a report to all the students' families in the form of a message. The transmitted message contains information such as the student's first name, last name, and school, e.g.,

```
*** Entering JenaSchoolSystemSemanticModelInterface.addStatement(Statement s) ...
*** ================================================================
*** Subject:   = http://www.ontologies.org/school#Christopher
*** Predicate: = http://austin.org/school#hasReport
*** Object:    = http://austin.org/school#True
*** Subject local name:   s = Christopher
*** Predicate local Name: s = hasReport
*** Object resource:      s = True
*** Object value = null
*** New school report : Christopher
message handlerMessage:
Type      = SCHOOL_REPORT
Source    = Pointers Run Elementary School
Destination    = Austin
Subject   = --- To: Austin family ...
---Message: Here's Christopher's Pointers Run Elementary School report ...
Body      = Report Form ...
========================================
First Name = Christopher
Last Name  = Austin
School Name = Pointers Run Elementary School
========================================
```

In this case, the mediator will match the message destination, Austin family, with the Austin family's semantic model interface, and forward the message.

Task 04. School Report Receipt. The Austin family's semantic model interface will identify the message type (i.e. school report), and trigger changes to the semantic model graph by adding a property "hasReportFrom" the elementary school to Christopher. The Austin family's semantic model interface is listening for changes to the semantic model graph. Once "hasReportFrom" is added to the graph, the

interface sends a message back to the elementary school to confirm the receipt of the report.

```
*** Entering JenaFamilySemanticModelInterface.addStatement(Statement s) ...
*** ========================================================================
*** Subject:   = http://www.ontologies.org/family#Christopher
*** Predicate: = http://austin.org/family#hasReportFrom
*** Object:    = Pointers Run Elementary School
*** Subject local name:   s = Christopher
*** Predicate local Name: s = hasReportFrom
*** Object resource:      s =
*** Node (Object) value = Pointers Run Elementary School
*** Object value = Pointers Run Elementary School
*** School Report was received for: Christopher
message handlerMessage:
Type      = SCHOOL_REPORT_RECEIVED
Source    = Austin
Destination    = Pointers Run Elementary School
Subject   = --- To: Pointers Run Elementary School ...
--- Message: The Austin family has received Christopher's
---          Pointers Run Elementary School report ...
Body = School Report Receipt Form ...
========================================
First Name  = Christopher
Last Name   = Austin
School Name = Pointers Run Elementary School
========================================
```

Again, the mediator will match the message destination, Pointers Run Elementary School, with the elementary school's semantic model interface, and forward the message. The elementary school's semantic model interface will identify the message type (i.e., school report received). In this case no further action is needed from the school side, so no changes to the semantic model graph are triggered.

4.2 Case Study 2: Family-School-Urban-Geography System Dynamics

In Case Study 1, we constructed a framework for modeling and simulating distributed system behavior that is affected by temporal events. In the behavior modeling of complex urban environments, however, notions of time and space are both critical to decision making. Case Study 2 builds upon the capabilities of Case Study 1 by capturing urban geography and introducing spatio-temporal reasoning into the behavior model.

In the model for family-school system interactions, temporal considerations include a child's age and events appearing on the school's academic calendar (e.g., enrollment period, school report period). Spatial considerations constrain the family-school system interactions further by only allowing enrollment of students who live within the school zone jurisdiction, and only providing school bus service to students who live beyond a certain distance from the school.

These determinations are done by comparing spatial entities, such as family addresses, school addresses, and school zone boundaries. Addresses are defined by latitude and longitude coordinates; therefore, a simple calculation using the latitudes and longitudes of two addresses can determine the distance between them. Similarly, school zones are defined by a collection of latitude and longitude coordinates that compose a polygon geometric shape. Any algorithm that solves the point-in-polygon (PIP) problem can determine if the address lies within the school

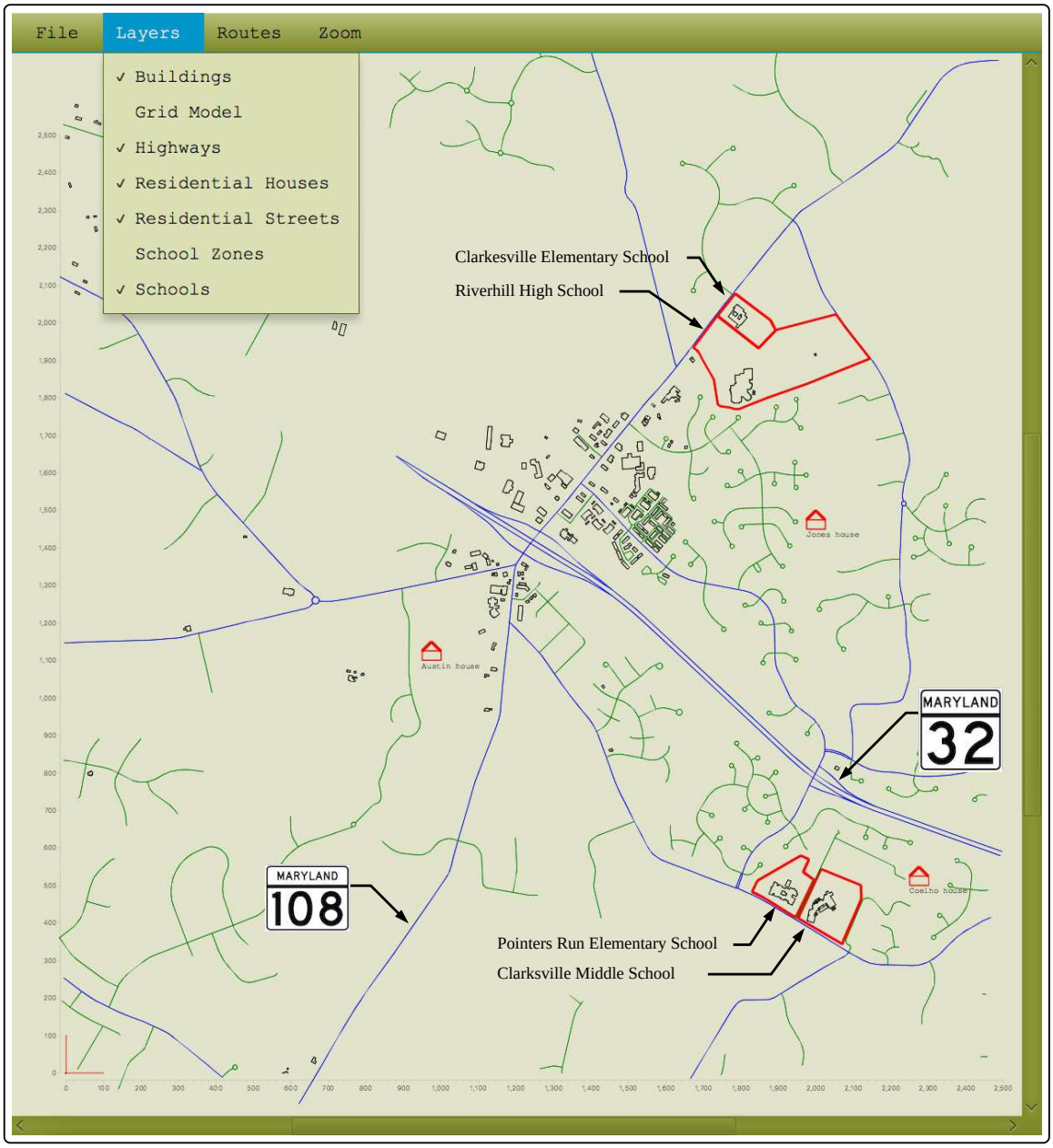

Figure 4.9: Schematic for schools in Columbia-Clarksville Area, Maryland, USA.

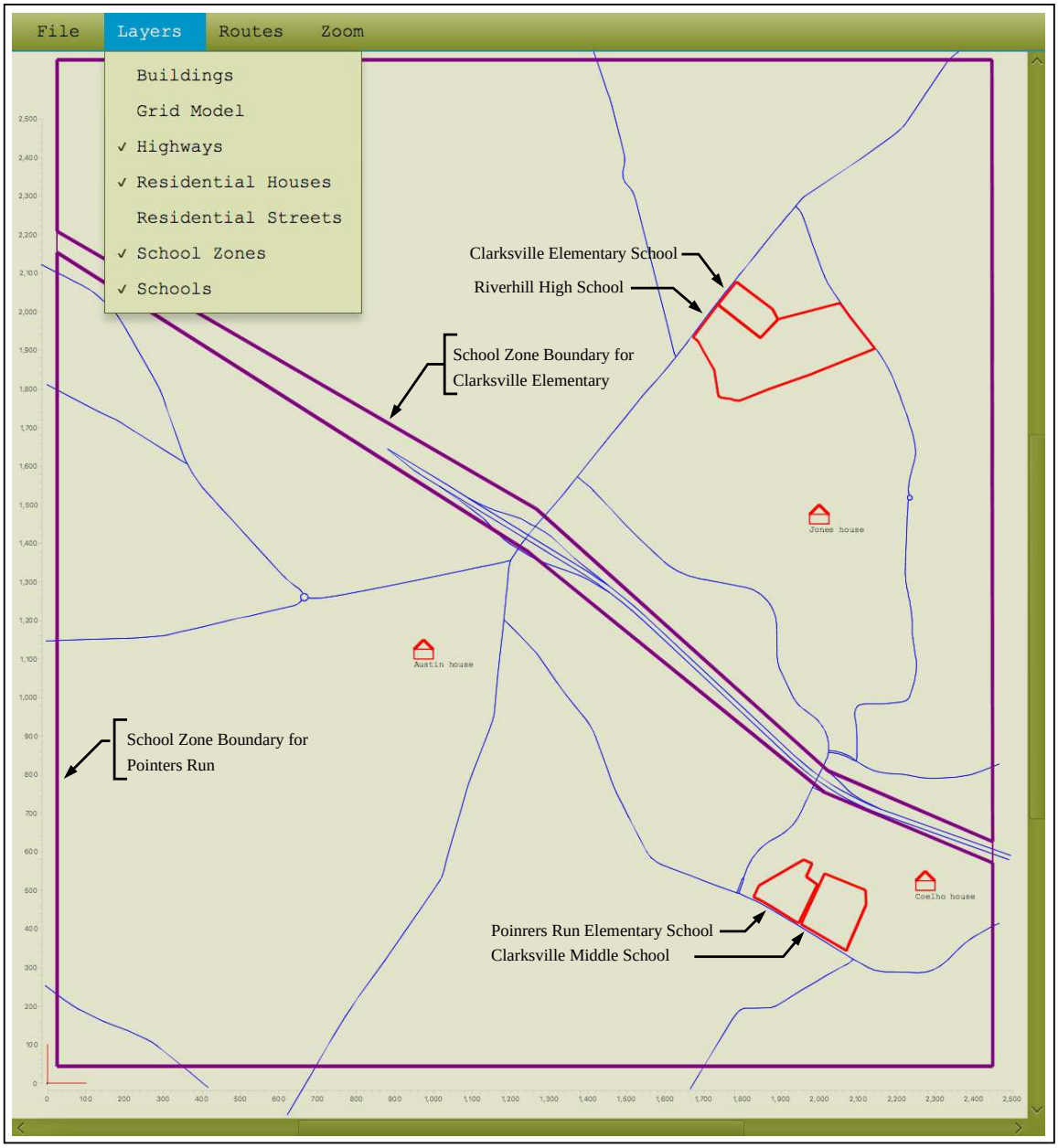

Figure 4.10: Elementary school zones in Columbia-Clarksville Area, Maryland, USA.

zone boundaries. This work uses OpenStreetMap tool to retrieve the latitudes and longitudes necessary for the these comparisons.

OpenStreetMap is a free, editable map of the whole world that is being built by volunteers and released with an open-content license. It has an exhaustive database of streets, cities, roads, buildings and so on, around the world. The real power of OpenStreetMap is the possibility to actually access the data behind its map rendering, different from others mapping providers [28].

In this case study, the urban area scenario for which an OpenstreetMap was retrieved is the city of Columbia,MD. As illustrated in Figure 4.9, the school district has one high school, one middle school and, in contrast to Case Study 1, two elementary schools. The presence of a second elementary school will allow modeling of decision making based on spatial constraints. We also introduce a third family that lives very close to one of the elementary schools in order to model the spatial constraint on the school bus service. Kids that live within a certain radius of the school can walk to class and have no right to the school bus service.

Figure 4.10 is a plan view of (fictitious) school zone boundaries in the Columbia-Clarksville Area, Maryland, USA. We assume that the middle and high schools will accept students from the entire region – their school zones are simply the rectangular shape. School zones for the elementary schools meet along Route 32. Students living North of Route 32 attend Clarksville Elementary. Students living South of Route 32 enroll in Pointers Run Elementary. The school zone polygons are defined in the OpenStreetMap input file.

4.2.1 Accessing Spatial Data from OpenStreetMap

OpenStreetMap provides the capability to query its database in various ways. In most cases, an XML file with descriptions of nodes (points of interest, facilities such as toilets, benches, addresses), ways (roads, water ways, transport routes) and areas (buildings, lakes) is available.

In our application, we have retrieved OpenStreet Map data for Columbia,MD in XML file format. Appendix E.1 contains an abbreviated description of the file. Schools and school zones are defined as ways, composed of a series of nodes, which in turn contain latitude and longitude information.

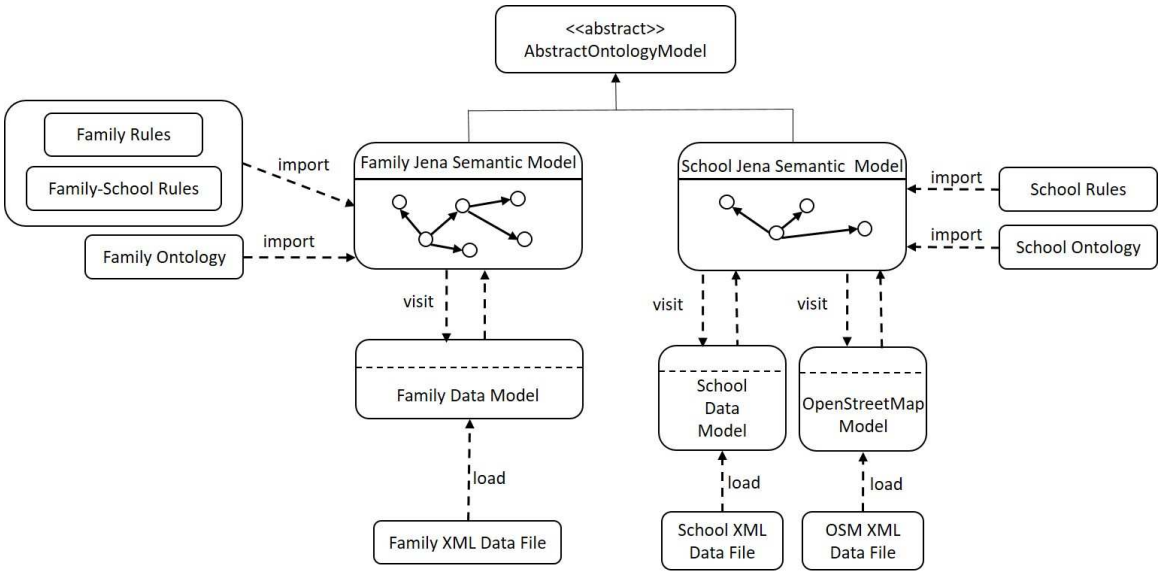

Figure 4.11: Generation of family and school semantic models, with input from the family data file, the school system data file, and data from OpenStreetMap.

In addition to the data retrieval process for the family XML datafile and school XML datafile described in Case Study 1, the OpenStreetMap XML datafile is parsed and imported into an OpenStreetMap Model, as shown in Figure 4.11. Then, we

create instances of the ontology spatial classes (i.e. Address), laden with the data from the OpenStreetMap XML file.

4.2.2 Extensions to the Family and School Ontologies

In addition to the family and school ontology definitions described in Case Study 1 and shown in Figures 4.3 and 4.4, certain ontology properties are added to the framework in order to allow modeling spatial behavior. The class Person in the family ontology now has property livesInSchoolZoneOf, and the class student in the school ontology now has properties livesInSchoolZoneOf and isElegibleForSchool-Bus. Descriptions of the extensions to the Case Study 1 OWL files for the family and school models are located in Appendix C.

Instantiating the Family and School Ontology Models. To instantiate these ontologies with information retrieved from the OpenStreetMap XML datafiles, a visitor object is used to visit the OpenStreetMap Model, retrieve the data, and create instances of ontology spatial classes with the data, as described in Figure 4.11. Similar to Case Study 1, the OpenStreetMap design assures visiting objects can only retrieve data pertaining to their corresponding ontology instance through a "password" mechanism. The data retrieved by the visitor object is then used to create instances of the ontology spatial classes in the Jena Semantic Models.

```
// Rules 02: Elementary school rules ...

[ EnterElementarySchool: (?x rdf:type af:Student) (?y rdf:type af:ElementarySchool)
     (?x af:livesInSchoolZoneOf ?y) (?x af:hasBirthDate ?a) getAge(?a,?b) ge(?b, 6)
     le(?b, 10) -> (?x af:attendsElementarySchool af:True) (?y af:hasStudent ?x)]

[ LeaveElementarySchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
     (?x af:attendsElementarySchool af:True) (?y af:hasStudent ?x)
     getAge(?a,?b) ge(?b, 10) -> remove(2) ]

// Rules 03: Middle school rules ...

[ EnterMiddleSchool: (?x rdf:type af:Student) (?y rdf:type af:MiddleSchool)
     (?x af:livesInSchoolZoneOf ?y) (?x af:hasBirthDate ?a) getAge(?a,?b) ge(?b, 11)
     le(?b, 13) -> (?x af:attendsMiddleSchool af:True) (?y af:hasStudent ?x) ]

[ LeaveMiddleSchool: (?x rdf:type af:Student) (?y rdf:type af:MiddleSchool)
     (?x af:hasBirthDate ?a) (?x af:attendsMiddleSchool af:True)
     (?y af:hasStudent ?x) getAge(?a,?b) ge(?b, 13) -> remove(2) ]

// Rules 04: High school rules ...

[ EnterHighSchool: (?x rdf:type af:Student) (?y rdf:type af:HighSchool)
     (?x af:livesInSchoolZoneOf ?y) (?x af:hasBirthDate ?a) getAge(?a,?b)
     ge(?b, 14) le(?b, 17) -> (?x af:attendsHighSchool af:True) (?y af:hasStudent ?x) ]

[ LeaveHighSchool: (?x rdf:type af:Student) (?y rdf:type af:HighSchool)
     (?x af:hasBirthDate ?a) (?x af:attendsHighSchool af:True) (?y af:hasStudent ?x)
     getAge(?a,?b) ge(?b, 17) -> remove(2) ]

// Rules 06: Elementary school transporation service rules ....

[ ElementarySchoolTransportationService: (?x rdf:type af:Student)
     (?y rdf:type af:ElementarySchool) (?y af:hasStudent ?x)
     (?x af:hasStudentAddress ?k) (?y af:hasSchoolAddress ?z)
     (?k af:hasLatitude ?l1) (?k af:hasLongitude ?l2)
     (?z af:hasLatitude ?l3) (?z af:hasLongitude ?l4) getDistance(?l1,?l2,?l3,?l4,?d)
     greaterThan(?d,1000)-> (?x af:isEligibleForSchoolBus af:True)  ]

... Middle school transportation rule removed ...

[ HighSchoolTransportationService: (?x rdf:type af:Student)
    (?y rdf:type af:HighSchool) (?y af:hasStudent ?x)
    (?x af:hasStudentAddress ?k) (?y af:hasSchoolAddress ?z)
    (?k af:hasLatitude ?l1) (?k af:hasLongitude ?l2) (?z af:hasLatitude ?l3)
    (?z af:hasLongitude ?l4) getDistance(?l1,?l2,?l3,?l4,?d)
    greaterThan(?d,2000)-> (?x af:isEligibleForSchoolBus af:True)  ]
```

Figure 4.12: Extended Jena rules for transformation of the School Semantic Model.

4.2.3 Extensions to the School Rules

Additional modifications and additions should be made to the school rules to account for spatial constraints. Descriptions of the rules extension for the school system model are presented in Figure 4.12.

School Rules. The combination of ontologies and ontology rules is also extremely powerful in scenarios where ontology graphs are space dependent. For example, consider the boy Christopher again, now identified as a student from the school system perspective. A built-in function getDistance() computes the the distance between Chistopher's address and the school address. A rule defined using Jena Rules determines whether or not Christopher is eligible to the school bus service by comparing the inputs to the built-in function getDistance(). If the output of the built-in is greater than a certain threshold distance, Christopher is entitled to the bus service, if not he will have to walk to school. In addition, school enrollment rules were modified to only allow students to enroll when they live within the school zone jurisdiction. Therefore, the graph transformations in the school system model can now occur due not only to input or time, but also space.

4.2.4 Assembly of the Family-School-Urban-Geography System

Since the step-by-step procedure for assembly and execution of the family-school-urban geography system is very similar to that of Case Study 1 (see Section 4.1.4), in this section we only provide details on ways in which Case Study 2 differs

from Case Study 1.

Step 01. Create semantic models; load ontologies. Case Study 2 has four schools (Riverhill Highschool; Clarksville Middle School; Pointers Run Elementary School; Clarksville Elementary School) and three families (Austin, Jones, and Coelho). The fragment of code:

```
JenaSchoolSystemSemanticModel rhs = new JenaSchoolSystemSemanticModel();
JenaSchoolSystemSemanticModel cms = new JenaSchoolSystemSemanticModel();
JenaSchoolSystemSemanticModel pes = new JenaSchoolSystemSemanticModel();
JenaSchoolSystemSemanticModel ces = new JenaSchoolSystemSemanticModel();

JenaFamilySemanticModel austin = new JenaFamilySemanticModel();
JenaFamilySemanticModel jones  = new JenaFamilySemanticModel();
JenaFamilySemanticModel coelho = new JenaFamilySemanticModel();
```

defines the semantic models and instantiates them with the appropriate school system and/or family ontologies.

Step 02. Connect family and school system models. The second step is to connect family and school semantic models via interfaces, and register each component with the mediator as message listeners for communication and information exchange with other components. The following fragment of code shows the minor extensions needed to include a third family (Coelho) and a second elementary school (Clarksville Elementary):

```
... Details of Austin and Jones family interface definitions removed ...

JenaFamilySemanticModelInterface   coelhofmi = coelho.getModelInterface();
coelhofmi.setType("Coelho");
```

```
... School definitions removed ...

JenaSchoolSystemSemanticModelInterface cesSmi = clarksvilleElementarySchool.getModelInterface
cesSmi.setType("Clarksville Elementary School");

// Coelho family will listen for messages from schools ...

coelhofmi.addMessageListener ( (MessageListener) rhsSmi );
coelhofmi.addMessageListener ( (MessageListener) cmsSmi );
coelhofmi.addMessageListener ( (MessageListener) pesSmi );
coelhofmi.addMessageListener ( (MessageListener) cesSmi );

// Clarksville Elementary will listen for messages from families ...

cesSmi.addMessageListener ( (MessageListener) austinfmi );
cesSmi.addMessageListener ( (MessageListener) jonesfmi );
cesSmi.addMessageListener ( (MessageListener) coelhofmi );
```

Step 03. Create visitor object models. Visitor objects are created for each semantic model, and the customized with password data. In Case Study 1 these objects visited the family and school system data models. Now, we also visit the OpenStreetMap data model to retrieve urban data relevant to the family and school system semantic models. The script of code:

```
FamilyDataModelJenaVisitor coelhoVisitor = new FamilyDataModelJenaVisitor();
coelhoVisitor.setPassword("Coelho");
coelhoVisitor.addSemanticModel( coelho );

SchoolSystemDataModelJenaVisitor cesVisitor = new SchoolSystemDataModelJenaVisitor();
cesVisitor.setPassword("Clarksville Elementary School");
cesVisitor.addSemanticModel( clarksvilleElementarySchool );

SchoolOSMJenaVisitor cesOSMvisitor = new SchoolOSMJenaVisitor();
cesOSMvisitor.setPassword("Clarksville Elementary School");
cesOSMvisitor.addSemanticModel( clarksvilleElementarySchool );
```

shows the essential details of setting up the visitor object models for the new family (Coelho) and new elementary school (Clarksville Elementary).

The following step is to create Data Models for the family and school ontologies, OpenStreetMap Models for the school ontologies, and populate them with data imported from XML datafiles.

Step 04. Get data from XML files. Only a few lines of Java are needed to instantiate the OpenStreetMap data model:

```
OpenStreetMapModel osm = new OpenStreetMapModel();

try{
   osm.getData( "data/columbia-school-district.osm");
} catch(Exception e){}
```

Step 05. Populate semantic models with individuals. We populate the semantic models with the data that visitors retrieve from the data models. The fragment of code:

```
fdm.accept ( coelhoVisitor );

ssdm.accept ( cesSchoolVisitor );
osm.accept ( cesOSMvisitor );
```

achieves three things. First, the semantic model for the Coelho family is populated with individuals, with data coming from the family data model. Next, the Clarksville

Elementary School is populated with individuals, with data coming from the School System Data Model. Finally, the semantic model for Clarksville Elementary School is augmented with data from the OpenStreetMap data model.

Step 06. Add rules. Finally, we import the rules into the semantic models and apply them.

```
ces.addRules ( "src/whistle/util/jena/rules/schoolRules.rules" );
ces.executeRules();

... details of adding rules to austin and jones families removed ...

coelho.addRules ( "src/whistle/util/jena/rules/familyRules.rules",
                  "src/whistle/util/jena/rules/familyschoolRules.rules" );
coelho.executeRules();
```

4.2.5 Simulation of Family-School-Urban-Geography Interactions

Task 01: Enrolling a Child in School. Just like in Case Study 1, when the family and family-school rules are added to the Austin family's semantic model, the graph of the Austin family may change. For example, the rules associated with age establish that Christopher Austin is now old enough to attend elementary school, and so the boolean property "attendsElementarySchool" becomes "True". The Austin family's semantic model interface will identify such a change. But now, instead of sending an enrollment request right away, the interface will send a request to the elementary schools in the area to check if the Austin family address is located within the their school zones. The request is sent in the form of a message,

containing relevant information such as Christopher's first name, last name, and address coordinates.

School enrollment requests are sent to two elementary schools:

```
[java] *** Entering JenaFamilySemanticModelInterface.addStatement(Statement s) ...
[java] *** ============================================================================
[java] *** Subject:   = http://www.ontologies.org/family#Christopher
[java] *** Predicate: = http://austin.org/family#attendsElementarySchool
[java] *** Object:    = http://austin.org/family#True
[java] *** Subject local name:   s = Christopher
[java] *** Predicate local Name: s = attendsElementarySchool
[java] *** Object resource:      s = True
[java] *** Object value = null
[java] *** Kid in school age: Christopher
[java] message handlerMessage:
[java] Type      = SCHOOL_ZONE_CHECK_REQUEST
[java] Source    = Austin
[java] Destination    = Pointers Run Elementary School
[java] Subject   = --- Request to check if this family lives in
                       your elementary school zone...
[java] Body      = SchoolZoneCheckRequestForm ...
[java] ========================================
[java]  First Name    = Christopher
[java]  Last Name   = Austin
[java]  Family Address Latitude  =  39.2013753
[java]  Family Address Longitude = -76.9498448
[java] ========================================
[java] message handlerMessage:
[java] Type      = SCHOOL_ZONE_CHECK_REQUEST
[java] Source    = Austin
[java] Destination    = Clarksville Elementary School
[java] Subject   = --- Request to check if this family lives in
                       your elementary school zone...
[java] Body      = SchoolZoneCheckRequestForm ...
[java] ========================================
[java]  First Name    = Christopher
[java]  Last Name   = Austin
[java]  Family Address Latitude  =  39.2013753
[java]  Family Address Longitude = -76.9498448
[java] ========================================
```

The mediator will match the messages destinations (i.e. Pointers Run and Clarksville Elementary School) with the corresponding elementary school's semantic model in-

terfaces, and forward the message. The elementary school semantic models interface will identify the message type (i.e. school zone check request), and decide whether the student lives within their school zone boundaries by comparing Christopher's address coordinates to the coordinates of points composing the school zone boundary. Since schools zones are unique and there is no overlap between different school zones, Chirstopher's address will be located inside of only one of the school zones. If it does, then a change to the school semantic model graph will be triggered by adding the property livesInSchoolZoneOf. The elementary school's semantic model interface is listening for changes to the semantic model graph. Once Christopher is identified as a potential student, the interface sends a message back to the Austin family to confirm his eligibility to enroll.

The process for school enrollment acceptance proceeds as follows:

```
*** Entering JenaSchoolSystemSemanticModelInterface.addStatement(Statement s) ...
*** ================================================================
*** Subject:   = http://www.ontologies.org/school#Christopher
*** Predicate: = http://austin.org/school#livesInSchoolZoneOf
*** Object:    = http://www.ontologies.org/school#Pointers Run Elementary School
*** Subject local name:   s = Christopher
*** Predicate local Name: s = livesInSchoolZoneOf
*** Object resource:      s = Pointers Run Elementary School
*** Object value = null
*** Please submit an enrollment form for : Christopher
message handlerMessage:
Type      = SCHOOL_ZONE_CONFIRMATION
Source    = Pointers Run Elementary School
Destination    = Austin
Subject   = --- To: Austin family ...
---Message: Pointers Run Elementary School is eligible to
            enroll with Pointers Run Elementary School...
Body      = SchoolZoneConfirmationForm ...
========================================
First Name = Christopher
Last Name  = Austin
School Name   = Pointers Run Elementary School
========================================
```

Again, the mediator will match the message destination, Austin family, with the Austin family's semantic model interface, and forward the message. The Austin family's semantic model interface will identify the message type (i.e., school zone confirmation). Then, a change to the family semantic model graph will be triggered by adding the property livesInSchoolZoneOf, this time associated with the family ontology. The Austin family semantic model interface is listening for changes to the semantic model graph. Once such change is identified, an enrollment request will be sent to the elementary school in the form of a message, containing relevant information such as Christopher's first name, last name, social security number and birth date.

From here on forward, the process continues as described in Case Study 1. Christopher will be registered as a student in the school. Rules associated with time will determine if it is time to send school reports home, and if it is, changes will be triggered in the school ontology graph. Similarly, rules associated with space will determine is Christopher is entitled to school bus service, and generate changes to the school ontology graph.

Chapter 5: **Conclusions and Future Work**

5.1 Summary and Conclusions

This paper has focused on the design and preliminary implementation of a message passing infrastructure needed to support communication in many-to-many association relationships connecting domain-specific networks. The long-term objective of this research is to build upon the family-school distributed behavior model and create models of the distributed behavior of urban infrastructure multi-level systems, and simulate cascading system failures that occur due to extreme external events. And we anticipate that the end-result will look something like Figure 1.4.

5.2 Future Work

We envision a full-scale implementation of distributed behavior modeling having to transmit a multiplicity of message types and content, with the underlying logic needed to deliver messages possibly being a lot more complicated than send message A in domain B to domain C. Our present-day capability is simplified in the sense that domain interfaces are assumed to be homogeneous and always working.

For problems involving recovery of services in an urban area after a disaster or attack, this will not always be true. This situation points to a strong need for new approaches to the construction and operation of message passing mechanisms.

One promising approach that we will explore is Apache Camel [20, 17], is an open source Java framework that focuses on making Enterprise Integration Patterns (EIP) accessible through carefully designed interfaces, the base objects, commonly needed implementations, debugging tools and a configuration system.

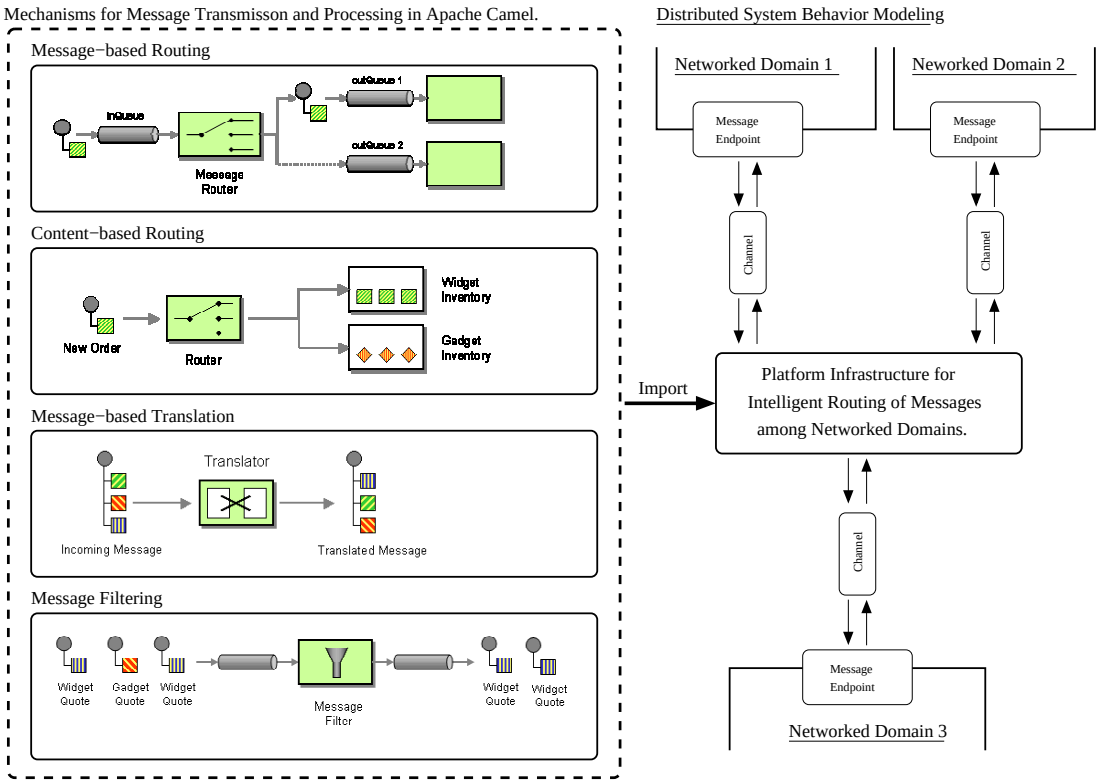

Figure 5.1: Platform infrastructure for distributed behavior modeling and intelligent communication (message passing) among networked domains.

Figure 5.1 shows, for example, a platform infrastructure for behavior modeling of three connected application (networked) domains. In addition to basic content-based routing, Apache Camel provides support for filtering and transformation of

messages.

A second important topic for future work is linkage of our simulation framework to tools for optimization and tradeoff analysis. Such tools would allow decision makers to examine the sensitivity of design outcomes to parameter choices, understand the impact of resource constraints, understand system stability in the presence of fluctuations to modeling parameter values, and potentially, even understand emergent interactions among systems.

Lastly, potential extension to the presented work, is in the development of ontologies. As it is presented in this work, the construction of ontologies is based on the data available from the XML datafiles, but this process is done manually. When modeling complex urban systems, this approach may become troublesome. A necessary step forward would be to implement natural language processing for the semi-automated identification of knowledge provided by the datafiles.

Appendix A: Model-Based Systems Engineering

A.1 Pathway from Operations Concept to Systems Design

Figure A.1 shows the pathway from an operations concept to simplified models for behavior and structure, requirements, system-level design and model checking. Because a system may not actually exist at this stage, the description will be written in the form of design requirements and mathematical constraints. Use cases are fragments of system functionality. A scenario is an example of typical system usage and describes the intended interaction between a system and its environment to achieve some purpose. Use cases and scenarios imply requirements, objects, and object interactions and interfaces in the stories they tell. Further design requirements and mathematical constraints can be obtained from the structure and communication of objects in the models for system functionality (e.g., required system interfaces). Models of behavior specify what the system will actually do. Models of structure specify how the system will accomplish its purpose. The system structure corresponds to collections of interconnected objects and subsystems, constrained by the environment within which the system must exist. The system-level design is created by mapping fragments of system functionality onto specific subsystems/objects in the system structure. Therefore, the behavior-to-structure mapping defines in a

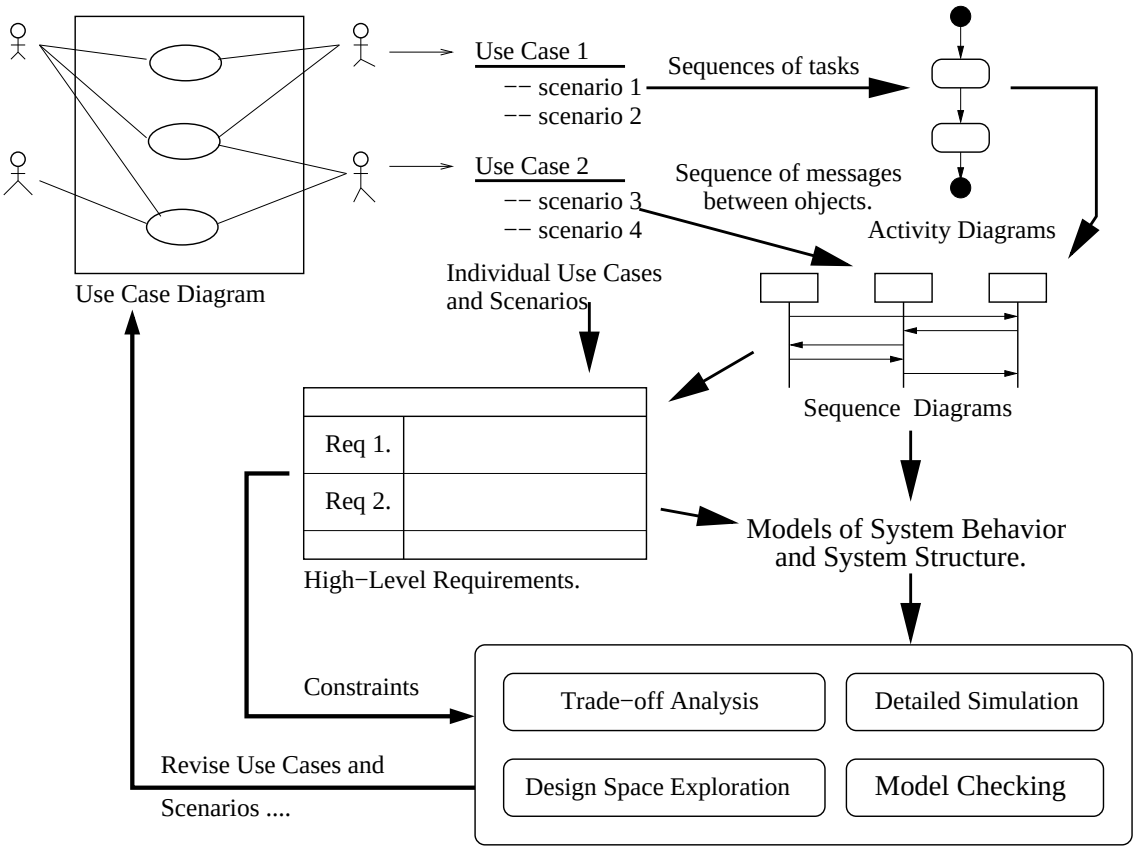

Figure A.1: Pathway from operations concept to simplified models for behavior and structure, to requirements, system-level design and model checking.

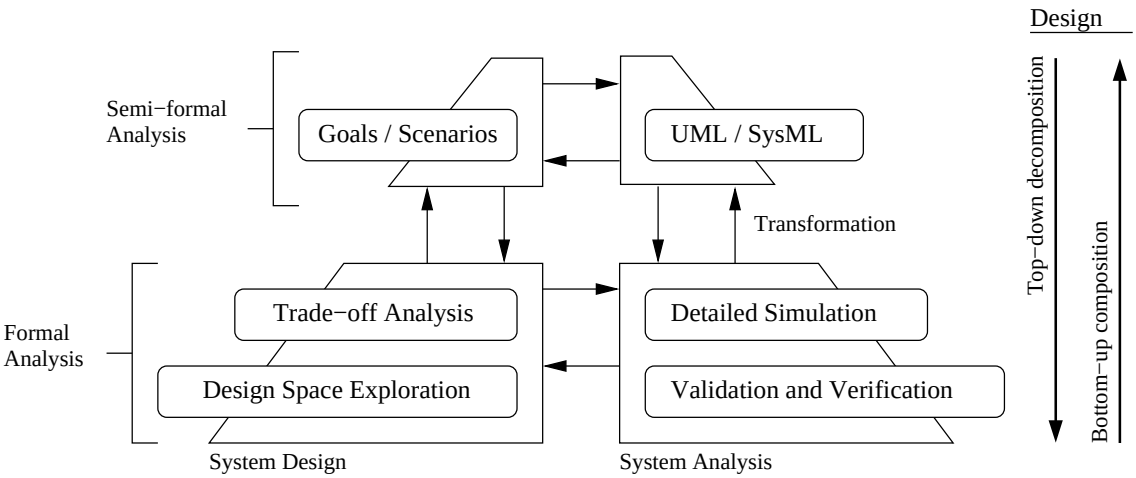

Figure A.2: Multi-level approach model-based systems engineering. Semi-formal models provide a high-level view of the complete system (efficiency). Formal models provide a detailed view of the actual system (accuracy)[25].

symbolic way the functional responsibility of each subsystem/component. Performance and characteristics of the system-level design are evaluated against the test requirements.

A.2 Strategies for dealing with Design Complexity

State-of-the-art practice in model-based systems engineering (MBSE) is to deal with design complexity through separation of concerns and development along disciplinary lines, followed by procedures for systems integration and validation and verification. While this approach eases work organization, design solutions tend to have loosely coupled system architectures that are limited in levels of achievable performance. Increases in system size and complexity drive the need for: (1) disciplined approaches to system design that involve the application of decomposition, composition, abstraction and use of semi-formal and formal analysis [2, 4, 21, 25], and (2) modeling formalisms that capture cause-and-effect relationships between designer concerns (e.g., correctness of system functionality; adequacy of performance; assurance of safety) and problem solutions.

In order to address these concerns, a multi-level approach to model-based system design must be taken. Figure A.2 describes the different levels of development to be used. The top level contains semi-formal models expressing ideas (i.e. goals and scenarios) and preliminary designs. Preliminary designs need to be represented by semi-formal models that have a fixed syntax and semantics, such as can be found in the System Modeling Language (SysML) [13]. Lower level models employ formal

languages having precisely defined semantics, and are designed to provide computational support for: (1) Detailed simulation of system behavior to evaluate levels of performance, (2) Validation and verification of the accuracy of functionality and control, (3) Systematic design space exploration, and (4) Trade-off analysis of design features.

Appendix B: Family and School System Data Models

This appendix contains complete descriptions of the family and school system models in XML. We employ JAXB technology to import the XML data files into the family and school system data models, respectively.

B.1 Family Data (FamilyModel.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<FamilyModel author="Maria Coelho" date="2017" source="UMD">

<Family>
    <attribute text="FamilyName" value="Austin"/>
    <attribute text="Address"
               value="6242 Heather Glen Way, Clarksville, MD 21029"/>
    <Person>
        <attribute text="Type" value="Male"/>
        <attribute text="FirstName" value="Mark"/>
        <attribute text="MiddleName" value="William"/>
        <attribute text="LastName" value="Austin"/>
        <attribute text="BirthDate" value="1704-06-10"/>
        <attribute text="Weight" value="170.0"/>
        <attribute text="Citizenship" value="New Zealand"/>
        <attribute text="SocialSecurity" value="111"/>
    </Person>
    <Person>
        <attribute text="Type" value="Boy"/>
        <attribute text="FirstName" value="Christopher"/>
        <attribute text="MiddleName" value="William"/>
        <attribute text="LastName" value="Austin"/>
        <attribute text="BirthDate" value="2007-10-12"/>
        <attribute text="Weight" value="40.0"/>
        <attribute text="Citizenship" value="New Zealand"/>
        <attribute text="SocialSecurity" value="678"/>
    </Person>
```

```
    <Person>
        <attribute text="Type" value="Female"/>
        <attribute text="FirstName" value="Nina"/>
        <attribute text="MiddleName" value="Marie"/>
        <attribute text="LastName" value="Austin"/>
        <attribute text="BirthDate" value="2005-12-25"/>
        <attribute text="Weight" value="42.0"/>
        <attribute text="Citizenship" value="New Zealand"/>
        <attribute text="Citizenship" value="US"/>
        <attribute text="SocialSecurity" value="345"/>
    </Person>
    <Person>
        <attribute text="Type" value="Female"/>
        <attribute text="FirstName" value="Erin"/>
        <attribute text="MiddleName" value="Mirrie"/>
        <attribute text="LastName" value="Austin"/>
        <attribute text="BirthDate" value="2012-01-12"/>
        <attribute text="Weight" value="22.0"/>
        <attribute text="Citizenship" value="US"/>
        <attribute text="SocialSecurity" value="123"/>
    </Person>
</Family>

<Family>
    <attribute text="FamilyName" value="Jones"/>
    <attribute text="Address"
               value="5807 Laurel Leaves Ln, Clarksville, MD 21029"/>
    <Person>
        <attribute text="Type" value="Male"/>
        <attribute text="FirstName" value="James"/>
        <attribute text="MiddleName" value="Robert"/>
        <attribute text="LastName" value="Jones"/>
        <attribute text="BirthDate" value="1972-06-10"/>
        <attribute text="Weight" value="180.0"/>
        <attribute text="Citizenship" value="England"/>
        <attribute text="SocialSecurity" value="222"/>
    </Person>
    <Person>
        <attribute text="Type" value="Boy"/>
        <attribute text="FirstName" value="Timothy"/>
        <attribute text="LastName" value="Jones"/>
        <attribute text="BirthDate" value="2007-10-12"/>
        <attribute text="Weight" value="43.0"/>
        <attribute text="Citizenship" value="US"/>
        <attribute text="Citizenship" value="England"/>
        <attribute text="SocialSecurity" value="1234"/>
    </Person>
    <Person>
        <attribute text="Type" value="Female"/>
        <attribute text="FirstName" value="Samantha"/>
        <attribute text="LastName" value="Jones"/>
        <attribute text="BirthDate" value="2004-12-25"/>
        <attribute text="Weight" value="41.0"/>
        <attribute text="Citizenship" value="US"/>
```

```
        <attribute text="Citizenship" value="England"/>
        <attribute text="SocialSecurity" value="3210"/>
    </Person>
</Family>
</FamilyModel>
```

Extensions for Case Study 2. The family XML datafile extensions for Case Study 2 are as follows:

```
<Family>
   <attribute text="FamilyName" value="Coelho"/>
   <Address>
       <attribute text="StreetAddress"
                  value="6724 Walter Scott Way, Columbia, MD 21044"/>
   </Address>
   <Person>
       <attribute text="Type" value="Male"/>
       <attribute text="FirstName" value="Luiz"/>
       <attribute text="MiddleName" value="Gonzaga"/>
       <attribute text="LastName" value="Coelho"/>
       <attribute text="BirthDate" value="1974-08-24"/>
       <attribute text="Weight" value="180.0"/>
       <attribute text="Citizenship" value="Brazil"/>
       <attribute text="SocialSecurity" value="456"/>
   </Person>
   <Person>
       <attribute text="Type" value="Female"/>
       <attribute text="FirstName" value="Maria"/>
       <attribute text="MiddleName" value="Eduarda"/>
       <attribute text="LastName" value="Coelho"/>
       <attribute text="BirthDate" value="2007-03-30"/>
       <attribute text="Weight" value="45.0"/>
       <attribute text="Citizenship" value="Brazil"/>
       <attribute text="SocialSecurity" value="393"/>
</Person>
</Family>
```

B.2 School System Data (SchoolSystemModel.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<SchoolSystemModel author="Maria Coelho" date="2017" source="UMD">
   <School>
       <attribute text="Type" value="High School"/>
       <attribute text="Name" value="River Hill High School"/>
       <attribute text="Grade" value="Grade09"/>
       <attribute text="Grade" value="Grade10"/>
       <attribute text="Grade" value="Grade11"/>
       <attribute text="Grade" value="Grade12"/>
       <attribute text="Report Period Start Time" value="2016-09-01T00:00:00"/>
       <attribute text="Report Period End Time" value="2020-10-20T00:00:00"/>
   </School>
   <School>
       <attribute text="Name" value="Clarksville Middle School"/>
       <attribute text="Type" value="Middle School"/>
       <attribute text="Grade" value="Grade06"/>
       <attribute text="Grade" value="Grade07"/>
       <attribute text="Grade" value="Grade08"/>
       <attribute text="Report Period Start Time" value="2016-09-01T00:00:00"/>
       <attribute text="Report Period End Time" value="2020-10-10T00:00:00"/>
   </School>
   <School>
       <attribute text="Name" value="Pointers Run Elementary School"/>
       <attribute text="Type" value="Elementary School"/>
       <attribute text="Grade" value="Grade01"/>
       <attribute text="Grade" value="Grade02"/>
       <attribute text="Grade" value="Grade03"/>
       <attribute text="Grade" value="Grade04"/>
       <attribute text="Grade" value="Grade05"/>
       <attribute text="Report Period Start Time" value="2016-09-01T00:00:00"/>
       <attribute text="Report Period End Time" value="2020-10-20T00:00:00"/>
   </School>
</SchoolSystemModel>
```

Extensions for Case Study 2. A second elementary school is added to the model:

```
<School>
    <attribute text="Name" value="Clarksville Elementary School"/>
    <attribute text="Type" value="Elementary School"/>
    <attribute text="Grade" value="Grade01"/>
    <attribute text="Grade" value="Grade02"/>
    <attribute text="Grade" value="Grade03"/>
    <attribute text="Grade" value="Grade04"/>
    <attribute text="Grade" value="Grade05"/>
    <attribute text="Report Period Start Time" value="2016-09-01T00:00:00"/>
    <attribute text="Report Period End Time" value="2020-10-20T00:00:00"/>
</School>
```

Appendix C: Family and School System Ontologies

This appendix contains complete descriptions of the family and school system ontologies written in OWL. The ontologies describe the knowledge (i.e., classes, data properties, and object properties), but not the data associated with specific individuals within each domain.

C.1 Family Ontology (umd-family.owl)

Family Ontology for Case Study 1. The family ontology for Case Study 1 has classes for Family, Person, Address, Male, Female, Boy, Child and Student.

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

<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY swrl "http://www.w3.org/2003/11/swrl#" >
    <!ENTITY swrlb "http://www.w3.org/2003/11/swrlb#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY owl2xml "http://www.w3.org/2006/12/owl2-xml#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
    <!ENTITY family_ontology "http://austin.org/family#" >
]>

<rdf:RDF xmlns="http://austin.org/family#"
     xml:base="http://austin.org/family"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:swrl="http://www.w3.org/2003/11/swrl#"
     xmlns:owl2xml="http://www.w3.org/2006/12/owl2-xml#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
```

```
 xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
 xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:family_ontology="http://austin.org/family#">
<owl:Ontology rdf:about="http://austin.org/family"/>

<!--
///////////////////////////////////////////////////////////////////
//
// Object Properties for class Family
//
///////////////////////////////////////////////////////////////////
 -->

<!-- http://austin.org/family#hasFamilyMember -->

<owl:ObjectProperty rdf:about="http://austin.org/family#hasFamilyMember">
    <rdfs:domain rdf:resource="http://austin.org/family#Family"/>
    <rdfs:range rdf:resource="http://austin.org/family#Person"/>
</owl:ObjectProperty>

<!-- http://austin.org/family#hasAddress -->

<owl:ObjectProperty rdf:about="http://austin.org/family#hasAddress">
    <rdfs:domain rdf:resource="http://austin.org/family#Family"/>
    <rdfs:range rdf:resource="http://austin.org/family#Address"/>
</owl:ObjectProperty>

<!--
///////////////////////////////////////////////////////////////////
//
// Object Properties for class Person
//
///////////////////////////////////////////////////////////////////
 -->

<!-- http://austin.org/family#belongsToFamily -->

<owl:ObjectProperty rdf:about="http://austin.org/family#belongsToFamily">
    <rdfs:domain rdf:resource="http://austin.org/family#Person"/>
    <rdfs:range rdf:resource="http://austin.org/family#Family"/>
</owl:ObjectProperty>

<!-- http://austin.org/family#hasSon -->

<owl:ObjectProperty rdf:about="http://austin.org/family#hasSon">
    <rdfs:range rdf:resource="http://austin.org/family#Person"/>
    <rdfs:domain rdf:resource="http://austin.org/family#Male"/>
</owl:ObjectProperty>

<!-- http://austin.org/family#hasDaughter -->

<owl:ObjectProperty rdf:about="http://austin.org/family#hasDaughter">
    <rdfs:range rdf:resource="http://austin.org/family#Person"/>
```

```
    <rdfs:domain rdf:resource="http://austin.org/family#Female"/>
</owl:ObjectProperty>

<!-- http://http://austin.org/family#hasFather -->

<owl:ObjectProperty rdf:about="http://austin.org/family#hasFather">
    <rdfs:domain rdf:resource="http://austin.org/family#Person"/>
</owl:ObjectProperty>

<!--
///////////////////////////////////////////////////////////////////
//
// Data properties for class Family
//
///////////////////////////////////////////////////////////////////
 -->

<!-- http://austin.org/family#hasFamilyName -->

<owl:DatatypeProperty rdf:about="http://austin.org/family#hasFamilyName">
    <rdfs:domain rdf:resource="http://austin.org/family#Family"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<!--
///////////////////////////////////////////////////////////////////
//
// Data properties for class Person
//
///////////////////////////////////////////////////////////////////
 -->

<!-- http://austin.org/family#hasFirstName -->

<owl:DatatypeProperty rdf:about="http://austin.org/family#hasFirstName">
    <rdfs:domain rdf:resource="http://austin.org/family#Person"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<!-- http://austin.org/family#hasLastName -->

<owl:DatatypeProperty rdf:about="http://austin.org/family#hasLastName">
    <rdfs:domain rdf:resource="http://austin.org/family#Person"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<!-- http://austin.org/family#hasSocialSecurityNo -->

<owl:DatatypeProperty rdf:about="http://austin.org/family#hasSocialSecurityNo">
    <rdfs:domain rdf:resource="http://austin.org/family#Person"/>
    <rdfs:range rdf:resource="&xsd;integer"/>
</owl:DatatypeProperty>

<!-- http://austin.org/family#hasAge -->
```

```
<owl:DatatypeProperty rdf:about="http://austin.org/family#hasAge">
    <rdfs:domain rdf:resource="http://austin.org/family#Person"/>
    <rdfs:range rdf:resource="&xsd;integer"/>
</owl:DatatypeProperty>

<!-- http://austin.org/family#hasWeight -->

<owl:DatatypeProperty rdf:about="http://austin.org/family#hasWeight">
    <rdfs:domain rdf:resource="http://austin.org/family#Person"/>
    <rdfs:range rdf:resource="&xsd;double"/>
</owl:DatatypeProperty>

<!-- http://austin.org/family#hasBirthDate -->

<owl:DatatypeProperty rdf:about="http://austin.org/family#hasBirthDate">
    <rdfs:domain rdf:resource="http://austin.org/family#Person"/>
    <rdfs:range rdf:resource="&xsd;date"/>
</owl:DatatypeProperty>

<!-- http://austin.org/family#hasCitizenship -->

<owl:DatatypeProperty rdf:about="http://austin.org/family#hasCitizenship">
    <rdfs:domain rdf:resource="http://austin.org/family#Person"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<!--
///////////////////////////////////////////////////////////////////
//
// Data properties for class Student
//
///////////////////////////////////////////////////////////////////
 -->

<!-- http://austin.org/family#attendsPreSchool -->

<owl:DatatypeProperty rdf:about="http://austin.org/family#attendsPreSchool">
    <rdfs:domain rdf:resource="http://austin.org/family#Student"/>
    <rdfs:range rdf:resource="&xsd;boolean"/>
</owl:DatatypeProperty>

<!-- http://austin.org/family#attendsSchool -->

<owl:DatatypeProperty rdf:about="http://austin.org/family#attendsSchool">
    <rdfs:domain rdf:resource="http://austin.org/family#Student"/>
    <rdfs:range rdf:resource="&xsd;boolean"/>
</owl:DatatypeProperty>

<!-- http://austin.org/family#attendsElementarySchool -->

<owl:DatatypeProperty rdf:about="http://austin.org/family#attendsElementarySchool">
    <rdfs:domain rdf:resource="http://austin.org/family#Student"/>
    <rdfs:range rdf:resource="&xsd;boolean"/>
```

```
</owl:DatatypeProperty>

<!-- http://austin.org/family#attendsMiddleSchool -->

<owl:DatatypeProperty rdf:about="http://austin.org/family#attendsMiddleSchool">
    <rdfs:domain rdf:resource="http://austin.org/family#Student"/>
    <rdfs:range rdf:resource="&xsd;boolean"/>
</owl:DatatypeProperty>

<!-- http://austin.org/family#attendsHighSchool -->

<owl:DatatypeProperty rdf:about="http://austin.org/family#attendsHighSchool">
    <rdfs:domain rdf:resource="http://austin.org/family#Student"/>
    <rdfs:range rdf:resource="&xsd;boolean"/>
</owl:DatatypeProperty>

<!-- http://austin.org/family#hasReportFrom -->

<owl:DatatypeProperty rdf:about="http://austin.org/family#hasReportFrom">
    <rdfs:domain rdf:resource="http://austin.org/family#Student"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<!--
///////////////////////////////////////////////////////////////////
//
// Data properties for class Address
//
///////////////////////////////////////////////////////////////////
 -->

<!-- http://austin.org/family#hasLatitude -->

<owl:DatatypeProperty rdf:about="http://austin.org/family#hasLatitude">
    <rdfs:domain rdf:resource="http://austin.org/family#Address"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<!-- http://austin.org/family#hasLongitude -->

<owl:DatatypeProperty rdf:about="http://austin.org/family#hasLongitude">
    <rdfs:domain rdf:resource="http://austin.org/family#Address"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<!--
///////////////////////////////////////////////////////////////////
//
// Classes
//
///////////////////////////////////////////////////////////////////
 -->

<!-- http://austin.org/family#Person -->
```

```
    <owl:Class rdf:about="http://austin.org/family#Person">
    </owl:Class>

    <!-- http://austin.org/family#Family -->

    <owl:Class rdf:about="http://austin.org/family#Family">
    </owl:Class>

    <!-- http://austin.org/family#Male -->

    <owl:Class rdf:about="http://austin.org/family#Male">
        <rdfs:subClassOf rdf:resource="http://austin.org/family#Person"/>
    </owl:Class>

    <!-- http://austin.org/family#Female -->

    <owl:Class rdf:about="http://austin.org/family#Female">
        <rdfs:subClassOf rdf:resource="http://austin.org/family#Person"/>
    </owl:Class>

    <!-- http://austin.org/family#Child -->

    <owl:Class rdf:about="http://austin.org/family#Child">
        <rdfs:subClassOf rdf:resource="http://austin.org/family#Person"/>
    </owl:Class>

    <!-- http://austin.org/family#Boy -->

    <owl:Class rdf:about="http://austin.org/family#Boy">
        <rdfs:subClassOf rdf:resource="http://austin.org/family#Male"/>
    </owl:Class>

    <!-- http://austin.org/family#Student -->

    <owl:Class rdf:about="http://austin.org/family#Student">
        <rdfs:subClassOf rdf:resource="http://austin.org/family#Person"/>
    </owl:Class>

    <!-- http://austin.org/family#Address -->

    <owl:Class rdf:about="http://austin.org/family#Address">
    </owl:Class>
</rdf:RDF>
<!-- Generated by the OWL API (version 3.5.0) http://owlapi.sourceforge.net -->
```

Extensions for Case Study 2. The family ontology extensions for Case Study 2 are as follows:

```
<!-- http://austin.org/family#livesInSchoolZoneOf -->

<owl:DatatypeProperty rdf:about="http://austin.org/family#livesInSchoolZoneOf">
    <rdfs:domain rdf:resource="http://austin.org/family#Student"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
```

C.2 School System Ontology (umd-school-system.owl)

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

<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY swrl "http://www.w3.org/2003/11/swrl#" >
    <!ENTITY swrlb "http://www.w3.org/2003/11/swrlb#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY owl2xml "http://www.w3.org/2006/12/owl2-xml#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
    <!ENTITY schoolsystem_ontology "http://austin.org/school#" >
]>

<rdf:RDF xmlns="http://austin.org/school#"
     xml:base="http://austin.org/school"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:swrl="http://www.w3.org/2003/11/swrl#"
     xmlns:owl2xml="http://www.w3.org/2006/12/owl2-xml#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:schoolsystem_ontology="http://austin.org/school#">
    <owl:Ontology rdf:about="http://austin.org/school"/>

    <!--
    ///////////////////////////////////////////////////////////////
    //
    // Object Properties for class School
    //
    ///////////////////////////////////////////////////////////////
     -->

    <!-- http://austin.org/school#hasGrade -->
```

```
<owl:ObjectProperty rdf:about="http://austin.org/school#hasGrade">
    <rdfs:domain rdf:resource="http://austin.org/school#School"/>
    <rdfs:range rdf:resource="http://austin.org/school#Grade"/>
</owl:ObjectProperty>

<!-- http://austin.org/school#hasStudent -->

<owl:ObjectProperty rdf:about="http://austin.org/school#hasStudent">
    <rdfs:domain rdf:resource="http://austin.org/school#School"/>
    <rdfs:range rdf:resource="http://austin.org/school#Student"/>
</owl:ObjectProperty>

<!-- http://austin.org/school#hasSchoolAddress -->

<owl:ObjectProperty rdf:about="http://austin.org/school#hasSchoolAddress">
    <rdfs:domain rdf:resource="http://austin.org/school#School"/>
    <rdfs:range rdf:resource="http://austin.org/school#Address"/>
</owl:ObjectProperty>

<!--
///////////////////////////////////////////////////////////////
//
// Object Properties for class Calendar
//
///////////////////////////////////////////////////////////////
 -->

<!-- http://austin.org/school#hasEvent -->

<owl:ObjectProperty rdf:about="http://austin.org/school#hasEvent">
    <rdfs:range rdf:resource="http://austin.org/school#Calendar"/>
    <rdfs:domain rdf:resource="http://austin.org/school#Event"/>
</owl:ObjectProperty>

<!--
///////////////////////////////////////////////////////////////
//
// Object Properties for class Student
//
///////////////////////////////////////////////////////////////
 -->

<!-- http://austin.org/school#isInGrade -->

<owl:ObjectProperty rdf:about="http://austin.org/school#isInGrade">
    <rdfs:range rdf:resource="http://austin.org/school#Student"/>
    <rdfs:domain rdf:resource="http://austin.org/school#Grade"/>
</owl:ObjectProperty>

<!-- http://austin.org/school#hasStudentAddress -->

<owl:ObjectProperty rdf:about="http://austin.org/school#hasStudentAddress">
    <rdfs:domain rdf:resource="http://austin.org/school#Student"/>
```

```
    <rdfs:range rdf:resource="http://austin.org/school#Address"/>
</owl:ObjectProperty>

<!--
///////////////////////////////////////////////////////////////
//
// Data properties for class School
//
///////////////////////////////////////////////////////////////
 -->

<!-- http://austin.org/school#hasName -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#hasName">
    <rdfs:domain rdf:resource="http://austin.org/school#School"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<!-- http://austin.org/school#hasEnrollment -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#hasEnrollment">
    <rdfs:domain rdf:resource="http://austin.org/school#Grade"/>
    <rdfs:range rdf:resource="&xsd;integer"/>
</owl:DatatypeProperty>

<!--
///////////////////////////////////////////////////////////////
//
// Data properties for class Student
//
///////////////////////////////////////////////////////////////
 -->

<!-- http://austin.org/school#hasFirstName -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#hasFirstName">
    <rdfs:domain rdf:resource="http://austin.org/school#Student"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<!-- http://austin.org/school#hasLastName -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#hasLastName">
    <rdfs:domain rdf:resource="http://austin.org/school#Student"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<!-- http://austin.org/school#hasSocialSecurityNo -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#hasSocialSecurityNo">
    <rdfs:domain rdf:resource="http://austin.org/school#Student"/>
    <rdfs:range rdf:resource="&xsd;integer"/>
</owl:DatatypeProperty>
```

```
<!-- http://austin.org/school#hasAge -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#hasAge">
    <rdfs:domain rdf:resource="http://austin.org/school#Student"/>
    <rdfs:range rdf:resource="&xsd;integer"/>
</owl:DatatypeProperty>

<!-- http://austin.org/school#hasBirthDate -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#hasBirthDate">
    <rdfs:domain rdf:resource="http://austin.org/school#Student"/>
    <rdfs:range rdf:resource="&xsd;date"/>
</owl:DatatypeProperty>

<!-- http://austin.org/school#attendsElementarySchool -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#attendsElementarySchool">
    <rdfs:domain rdf:resource="http://austin.org/school#Student"/>
    <rdfs:range rdf:resource="&xsd;boolean"/>
</owl:DatatypeProperty>

<!-- http://austin.org/school#attendsMiddleSchool -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#attendsMiddleSchool">
    <rdfs:domain rdf:resource="http://austin.org/school#Student"/>
    <rdfs:range rdf:resource="&xsd;boolean"/>
</owl:DatatypeProperty>

<!-- http://austin.org/school#attendsHighSchool -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#attendsHighSchool">
    <rdfs:domain rdf:resource="http://austin.org/school#Student"/>
    <rdfs:range rdf:resource="&xsd;boolean"/>
</owl:DatatypeProperty>

<!-- http://austin.org/school#hasReport -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#hasReport">
    <rdfs:domain rdf:resource="http://austin.org/school#Student"/>
    <rdfs:range rdf:resource="&xsd;boolean"/>
</owl:DatatypeProperty>

<!--
///////////////////////////////////////////////////////////////////
//
// Data properties for class Event
//
///////////////////////////////////////////////////////////////////
 -->

<!-- http://austin.org/school#hasStartTime -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#hasStartTime">
    <rdfs:domain rdf:resource="http://austin.org/school#Event"/>
```

```
    <rdfs:range rdf:resource="&xsd;dateTime"/>
</owl:DatatypeProperty>

<!-- http://austin.org/school#hasEndTime -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#hasEndTime">
    <rdfs:domain rdf:resource="http://austin.org/school#Event"/>
    <rdfs:range rdf:resource="&xsd;dateTime"/>
</owl:DatatypeProperty>

<!--
///////////////////////////////////////////////////////////////
//
// Data properties for class Address
//
///////////////////////////////////////////////////////////////
 -->

<!-- http://austin.org/school#hasLatitude -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#hasLatitude">
    <rdfs:domain rdf:resource="http://austin.org/school#Address"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<!-- http://austin.org/school#hasLongitude -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#hasLongitude">
    <rdfs:domain rdf:resource="http://austin.org/school#Address"/>
    <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>

<!--
///////////////////////////////////////////////////////////////
//
// Classes
//
///////////////////////////////////////////////////////////////
 -->

<!-- http://austin.org/school#Student -->

<owl:Class rdf:about="http://austin.org/school#Student">
</owl:Class>

<!-- http://austin.org/school#School -->

<owl:Class rdf:about="http://austin.org/school#School">
</owl:Class>

<!-- http://austin.org/school#Grade -->

<owl:Class rdf:about="http://austin.org/school#Grade">
</owl:Class>
```

```
<!-- http://austin.org/school#Calendar -->

<owl:Class rdf:about="http://austin.org/school#Calendar">
</owl:Class>

<!-- http://austin.org/school#Event -->

<owl:Class rdf:about="http://austin.org/school#Event">
</owl:Class>

<!-- http://austin.org/school#HighSchool -->

<owl:Class rdf:about="http://austin.org/school#HighSchool">
    <rdfs:subClassOf rdf:resource="http://austin.org/school#School"/>
</owl:Class>

<!-- http://austin.org/school#MiddleSchool -->

<owl:Class rdf:about="http://austin.org/school#MiddleSchool">
    <rdfs:subClassOf rdf:resource="http://austin.org/school#School"/>
</owl:Class>

<!-- http://austin.org/school#ElementarySchool -->

<owl:Class rdf:about="http://austin.org/school#ElementarySchool">
    <rdfs:subClassOf rdf:resource="http://austin.org/school#School"/>
</owl:Class>

<!-- http://austin.org/school#Grade01 -->

<owl:Class rdf:about="http://austin.org/school#Grade01">
    <rdfs:subClassOf rdf:resource="http://austin.org/school#Grade"/>
</owl:Class>

<!-- http://austin.org/school#Grade02 -->

<owl:Class rdf:about="http://austin.org/school#Grade02">
    <rdfs:subClassOf rdf:resource="http://austin.org/school#Grade"/>
</owl:Class>

<!-- http://austin.org/school#Grade03 -->

<owl:Class rdf:about="http://austin.org/school#Grade03">
    <rdfs:subClassOf rdf:resource="http://austin.org/school#Grade"/>
</owl:Class>

<!-- http://austin.org/school#Grade04 -->

<owl:Class rdf:about="http://austin.org/school#Grade04">
    <rdfs:subClassOf rdf:resource="http://austin.org/school#Grade"/>
</owl:Class>

<!-- http://austin.org/school#Grade05 -->
```

```
    <owl:Class rdf:about="http://austin.org/school#Grade05">
        <rdfs:subClassOf rdf:resource="http://austin.org/school#Grade"/>
    </owl:Class>

    <!-- http://austin.org/school#Grade06 -->

    <owl:Class rdf:about="http://austin.org/school#Grade06">
        <rdfs:subClassOf rdf:resource="http://austin.org/school#Grade"/>
    </owl:Class>

    <!-- http://austin.org/school#Grade07 -->

    <owl:Class rdf:about="http://austin.org/school#Grade07">
        <rdfs:subClassOf rdf:resource="http://austin.org/school#Grade"/>
    </owl:Class>

    <!-- http://austin.org/school#Grade08 -->

    <owl:Class rdf:about="http://austin.org/school#Grade08">
        <rdfs:subClassOf rdf:resource="http://austin.org/school#Grade"/>
    </owl:Class>

    <!-- http://austin.org/school#Grade09 -->

    <owl:Class rdf:about="http://austin.org/school#Grade09">
        <rdfs:subClassOf rdf:resource="http://austin.org/school#Grade"/>
    </owl:Class>

    <!-- http://austin.org/school#Grade10 -->

    <owl:Class rdf:about="http://austin.org/school#Grade10">
        <rdfs:subClassOf rdf:resource="http://austin.org/school#Grade"/>
    </owl:Class>

    <!-- http://austin.org/school#Grade11 -->

    <owl:Class rdf:about="http://austin.org/school#Grade11">
        <rdfs:subClassOf rdf:resource="http://austin.org/school#Grade"/>
    </owl:Class>

    <!-- http://austin.org/school#Grade12 -->

    <owl:Class rdf:about="http://austin.org/school#Grade12">
        <rdfs:subClassOf rdf:resource="http://austin.org/school#Grade"/>
    </owl:Class>

    <!-- http://austin.org/school#Address -->

    <owl:Class rdf:about="http://austin.org/school#Address">
    </owl:Class>
</rdf:RDF>
```

Extensions for Case Study 2. The school ontology extensions for Case Study 2 are as follows:

```
<!-- http://austin.org/school#livesInSchoolZoneOf -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#livesInSchoolZoneOf">
    <rdfs:domain rdf:resource="http://austin.org/school#Student"/>
    <rdfs:range rdf:resource="http://austin.org/school#School"/>
</owl:DatatypeProperty>

<!-- http://austin.org/school#hasSchoolZoneBoundaryPt -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#hasSchoolZoneBoundaryPt">
    <rdfs:domain rdf:resource="http://austin.org/school#School"/>
    <rdfs:range rdf:resource="&xsd;string[]"/>
</owl:DatatypeProperty>

<!-- http://austin.org/school#isElegibleForSchoolBus -->

<owl:DatatypeProperty rdf:about="http://austin.org/school#elegibleForSchoolBus">
    <rdfs:domain rdf:resource="http://austin.org/school#Student"/>
    <rdfs:range rdf:resource="&xsd;boolean"/>
</owl:DatatypeProperty>
```

Appendix D: Family and School System Rules

This appendix contains complete descriptions of: (1) the family rules, (2) the school system rules, and (3) the family-school interaction rules. After these rule files have been imported into Jena, the Jena Reasoner transforms the Semantic Graph in response to events (e.g., an incoming message).

D.1 Family Rules (umd-family.rules)

Family Rules for Case Study 1. This set of rules identifies relationships and properties within a family.

```
@prefix af:  <http://austin.org/family#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

// Rule 01: Propagate class hierarchy relationships ....

[ rdfs01: (?x rdfs:subClassOf ?y), notEqual(?x,?y),
          (?a rdf:type ?x) -> (?a rdf:type ?y)]

// Rule 02: Family rules ....

[ Family: (?x rdf:type af:Family) (?x af:hasFamilyMember ?y) ->
          (?y af:belongsToFamily ?x) ]

// Rule 02: Identify a person who is also a child ...

[ Child:  (?x rdf:type af:Person) (?x af:hasAge ?y)
          lessThan(?y, 18) -> (?x rdf:type af:Child) ]

[ UpdateChild: (?x rdf:type af:Child) (?x af:hasBirthDate ?y)
```

```
          getAge(?y,?b) ge(?b, 18) -> remove(0) ]

// Rule 03: Identify a person who is also a student ...

[ Student: (?x rdf:type af:Person) (?x af:hasAge ?y) greaterThan(?y, 4)
           lessThan(?y, 18) -> (?x rdf:type af:Student) ]

[ UpdateStudent: (?x rdf:type af:Student) (?x af:hasBirthDate ?y)
           getAge(?y,?b) ge(?b, 18) -> remove(0) ]

// Rule 04: Compute and store the age of a person ....

[ GetAge:  (?x rdf:type af:Person) (?x af:hasBirthDate ?y)
           getAge(?y,?z) -> (?x af:hasAge ?z) ]

[ UpdateAge: (?a rdf:type af:Person) (?a af:hasBirthDate ?b)
             (?a af:hasAge ?c) getAge(?b,?d) notEqual(?c, ?d) ->
             remove(2) (?a af:hasAge ?d) ]

// Rule 05: Set father-son and father-daughter relationships ...

[ SetFather01: (?f rdf:type af:Male) (?f af:hasSon ?s)-> (?s af:hasFather ?f)]
[ SetFather02: (?f rdf:type af:Male) (?f af:hasDaughter ?s)-> (?s af:hasFather ?f)]
```

D.2 School System Rules (umd-school-system.rules)

School System Rules for Case Study 1. Rules are provided for attendance, progression through the grades and timing of school reports.

```
@prefix af:  <http://austin.org/school#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

// Rule 01: Propagate class hierarchy relationships ....

[ rdfs01: (?x rdfs:subClassOf ?y), notEqual(?x,?y),
          (?a rdf:type ?x) -> (?a rdf:type ?y)]

// Rules 02: Elementary school rules ...

[ EnterElementarySchool: (?x rdf:type af:Student)
     (?y rdf:type af:ElementarySchool) (?x af:hasBirthDate ?a)
     getAge(?a,?b) ge(?b, 6) le(?b, 10) ->
     (?x af:attendsElementarySchool af:True) (?y af:hasStudent ?x)]
```

```
[ LeaveElementarySchool: (?x rdf:type af:Student)
     (?x af:hasBirthDate ?a) (?x af:attendsElementarySchool af:True)
     (?y af:hasStudent ?x) getAge(?a,?b) ge(?b, 10) -> remove(2) ]

[ GradeOne:   (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
              getAge(?a,?b) equal(?b, 6) -> (?x af:isInGrade af:Grade01) ]
[ GradeTwo:   (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
              getAge(?a,?b) equal(?b, 7) -> (?x af:isInGrade af:Grade02) ]
[ GradeThree: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
              getAge(?a,?b) equal(?b, 8) -> (?x af:isInGrade af:Grade03) ]
[ GradeFour:  (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
              getAge(?a,?b) equal(?b, 9) -> (?x af:isInGrade af:Grade04) ]
[ GradeFive:  (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
              getAge(?a,?b) equal(?b, 10) -> (?x af:isInGrade af:Grade05)]

// Rules 03: Middle school rules ...

[ EnterMiddleSchool: (?x rdf:type af:Student)
     (?y rdf:type af:MiddleSchool) (?x af:hasBirthDate ?a)
     getAge(?a,?b) ge(?b, 11) le(?b, 13) ->
     (?x af:attendsMiddleSchool af:True) (?y af:hasStudent ?x) ]

[ LeaveMiddleSchool: (?x rdf:type af:Student) (?y rdf:type af:MiddleSchool)
     (?x af:hasBirthDate ?a) (?x af:attendsMiddleSchool af:True)
     (?y af:hasStudent ?x) getAge(?a,?b) ge(?b, 13) -> remove(2) ]

[ GradeSix:   (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
              getAge(?a,?b) equal(?b, 11) -> (?x af:isInGrade af:Grade06) ]

[ GradeSeven: (?x rdf:type af:Student)  (?x af:hasBirthDate ?a)
              getAge(?a,?b) equal(?b, 12) -> (?x af:isInGrade af:Grade07) ]

[ GradeEight: (?x rdf:type af:Student)  (?x af:hasBirthDate ?a)
              getAge(?a,?b) equal(?b, 13) -> (?x af:isInGrade af:Grade08) ]

// Rules 04: High school rules ...

[ EnterHighSchool: (?x rdf:type af:Student)
     (?y rdf:type af:HighSchool) (?x af:hasBirthDate ?a)
     getAge(?a,?b) ge(?b, 14) le(?b, 17) ->
     (?x af:attendsHighSchool af:True) (?y af:hasStudent ?x) ]

[ LeaveHighSchool: (?x rdf:type af:Student) (?y rdf:type af:HighSchool)
     (?x af:hasBirthDate ?a) (?x af:attendsHighSchool af:True)
     (?y af:hasStudent ?x) getAge(?a,?b) ge(?b, 17) -> remove(2) ]

[ GradeNine: (?x rdf:type af:Student)  (?x af:hasBirthDate ?a)
     getAge(?a,?b) equal(?b, 14) -> (?x af:isInGrade af:Grade09) ]
[ GradeTen:  (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
     getAge(?a,?b) equal(?b, 15) -> (?x af:isInGrade af:Grade10) ]
[ GradeEleven: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
     getAge(?a,?b) equal(?b, 16) -> (?x af:isInGrade af:Grade11) ]
[ GradeTwelve: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
```

```
     getAge(?a,?b) equal(?b, 17) -> (?x af:isInGrade af:Grade12) ]

// Rules 05: If today is report period, send school report ....

[ GenerateReport: (?x rdf:type af:Event) (?y rdf:type af:Student)
     (?z rdf:type af:School)  (?z af:hasStudent ?y) (?x af:hasStartTime ?t1)
     (?x af:hasEndTime ?t2) getToday(?t3) lessThan(?t3,?t2)
     greaterThan(?t3,?t1) -> (?y af:hasReport af:True)  ]
```

Extensions for Case Study 2. The school rules are extended so that a child will only be admitted to the school if they fall within the acceptable age range (temporal), and their home address is within the school zone (spatial).

```
// Rules 02: Elementary school rules ...

[ EnterElementarySchool: (?x rdf:type af:Student)
     (?y rdf:type af:ElementarySchool) (?x af:livesInSchoolZoneOf ?y)
     (?x af:hasBirthDate ?a) getAge(?a,?b) ge(?b, 6) le(?b, 10) ->
     (?x af:attendsElementarySchool af:True) (?y af:hasStudent ?x)]

[ LeaveElementarySchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
     (?x af:attendsElementarySchool af:True) (?y af:hasStudent ?x)
     getAge(?a,?b) ge(?b, 10) -> remove(2) ]

// Rules 03: Middle school rules ...

[ EnterMiddleSchool: (?x rdf:type af:Student) (?y rdf:type af:MiddleSchool)
     (?x af:livesInSchoolZoneOf ?y) (?x af:hasBirthDate ?a) getAge(?a,?b)
     ge(?b, 11) le(?b, 13) -> (?x af:attendsMiddleSchool af:True) (?y af:hasStudent ?x) ]

[ LeaveMiddleSchool: (?x rdf:type af:Student) (?y rdf:type af:MiddleSchool)
     (?x af:hasBirthDate ?a) (?x af:attendsMiddleSchool af:True) (?y af:hasStudent ?x)
     getAge(?a,?b) ge(?b, 13) -> remove(2) ]

// Rules 04: High school rules ...

[ EnterHighSchool: (?x rdf:type af:Student) (?y rdf:type af:HighSchool)
     (?x af:livesInSchoolZoneOf ?y) (?x af:hasBirthDate ?a) getAge(?a,?b)
     ge(?b, 14) le(?b, 17) -> (?x af:attendsHighSchool af:True) (?y af:hasStudent ?x) ]

[ LeaveHighSchool: (?x rdf:type af:Student) (?y rdf:type af:HighSchool)
     (?x af:hasBirthDate ?a) (?x af:attendsHighSchool af:True)
     (?y af:hasStudent ?x) getAge(?a,?b) ge(?b, 17) -> remove(2) ]

// Rules 06: School transporation service rules ....
```

```
[ ElementarySchoolTransportationService: (?x rdf:type af:Student)
     (?y rdf:type af:ElementarySchool) (?y af:hasStudent ?x)
     (?x af:hasStudentAddress ?k) (?y af:hasSchoolAddress ?z)
     (?k af:hasLatitude ?l1) (?k af:hasLongitude ?l2) (?z af:hasLatitude ?l3)
     (?z af:hasLongitude ?l4) getDistance(?l1,?l2,?l3,?l4,?d)
     greaterThan(?d,1000) -> (?x af:isElegibleForSchoolBus af:True)  ]

[ MiddleSchoolTransportationService: (?x rdf:type af:Student)
     (?y rdf:type af:MiddleSchool) (?y af:hasStudent ?x)
     (?x af:hasStudentAddress ?k) (?y af:hasSchoolAddress ?z)
     (?k af:hasLatitude ?l1) (?k af:hasLongitude ?l2) (?z af:hasLatitude ?l3)
     (?z af:hasLongitude ?l4) getDistance(?l1,?l2,?l3,?l4,?d)
     greaterThan(?d,1500)-> (?x af:isElegibleForSchoolBus af:True)  ]

[ HighSchoolTransportationService: (?x rdf:type af:Student)
     (?y rdf:type af:HighSchool) (?y af:hasStudent ?x) (?x af:hasStudentAddress ?k)
     (?y af:hasSchoolAddress ?z) (?k af:hasLatitude ?l1) (?k af:hasLongitude ?l2)
     (?z af:hasLatitude ?l3) (?z af:hasLongitude ?l4) getDistance(?l1,?l2,?l3,?l4,?d)
     greaterThan(?d,2000)-> (?x af:isElegibleForSchoolBus af:True)  ]
```

D.3 School-Family Interaction Rules (umd-school-family-interaction.rules)

Here's what a family needs to know about school requirements:

```
// ================================================================
// School-family interaction rules ...
// ================================================================

@prefix af:  <http://austin.org/family#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

// Rules 01: Children of age 4 and 5 attend preschool ...

[ EnterPreSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
     getAge(?a,?b) ge(?b, 4) le(?b, 5) -> (?x af:attendsPreSchool af:True) ]

[ LeavePreSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
     (?x af:attendsPreSchool af:True) getAge(?a,?b) ge(?b, 6) -> remove(2) ]

// Rules 02: Children aged 6 through 10 attend elementary school ....

[ EnterSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
     getAge(?a,?b) ge(?b, 6) le(?b, 10) -> (?x af:attendsElementarySchool af:True) ]
```

```
[ LeaveSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
     (?x af:attendsElementarySchool af:True) getAge(?a,?b) ge(?b, 11) -> remove(2) ]

// Rules 03: Children aged 11 through 13 attend middle school ....

[ EnterSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
     getAge(?a,?b) ge(?b, 11) le(?b, 13) -> (?x af:attendsMiddleSchool af:True) ]

[ LeaveSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
     (?x af:attendsMiddleSchool af:True) getAge(?a,?b) ge(?b, 14) -> remove(2) ]

// Rules 04: Children aged 14 through 17 attend high school ....

[ EnterSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
     getAge(?a,?b) ge(?b, 14) le(?b, 17) -> (?x af:attendsHighSchool af:True) ]

[ LeaveSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
     (?x af:attendsHighSchool af:True) getAge(?a,?b) ge(?b, 18) -> remove(2) ]

// Rules 05: Children aged 6 through 18 attend regular school ....

[ EnterSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
     getAge(?a,?b) ge(?b, 6) le(?b, 17) -> (?x af:attendsSchool af:True) ]

[ LeaveSchool: (?x rdf:type af:Student) (?x af:hasBirthDate ?a)
     (?x af:attendsSchool af:True) getAge(?a,?b) ge(?b, 18) -> remove(2) ]
```

Appendix E: OpenStreetMap Data for Columbia, MD

This appendix contains an abbreviated description of the OpenStreeMap (OSM) for Columbia, Maryland. The complete data file is 79,500 lines long – not large by OSM standards, but still more than 1,000 pages of text – so we present only the essential details defining the public elementary, middle and high schools in the area, and the two main highways (MD Routes 32 and 108).

E.1 OpenStreetMap Data File (columbia-school-district.osm)

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" copyright="OpenStreetMap and contributors"
     attribution="http://www.openstreetmap.org/copyright"
     license="http://opendatacommons.org/licenses/odbl/1-0/">

<bounds minlat="39.1822000" minlon="-76.9706000"
        maxlat="39.2398000" maxlon="-76.9139000"/>

 <node id="31843411" visible="true" version="3" changeset="7939286"
      timestamp="2011-04-22T22:46:47Z" user="asciiphil" uid="247807"
      lat="39.1844418" lon="-76.8964387"/>

 <node id="34031823" visible="true" version="4" changeset="10355745"
      timestamp="2012-01-10T21:39:40Z" user="asciiphil" uid="247807"
      lat="39.2065197" lon="-76.9335257"/>

 ... nodes deleted ...

 <node id="37016495" visible="true" version="3" changeset="12600093"
       timestamp="2012-08-03T16:15:56Z" user="asciiphil" uid="247807"
       lat="39.1982520" lon="-76.9659773">
       <tag k="highway" v="turning_circle"/>
```

```
</node>

... nodes deleted ...

<node id="358240259" visible="true" version="1" changeset="774950"
     timestamp="2009-03-10T03:40:11Z" user="iandees" uid="4732"
     lat="39.2053843" lon="-76.9433093">
     <tag k="amenity" v="school"/>
     <tag k="width" v="6"/>
     <tag k="ele"   v="156"/>
     <tag k="name" v="Saint Louis School"/>
</node>

... nodes deleted ...

<node id="358255578" visible="true" version="1" changeset="774950"
     timestamp="2009-03-10T04:30:26Z" user="iandees" uid="4732"
     lat="39.2065147" lon="-76.9450387">
     <tag k="amenity" v="school"/>
     <tag k="width" v="6"/>
     <tag k="ele" v="146"/>
     <tag k="gnis:state_id" v="24"/>
     <tag k="name" v="Brookfield Christian Elementary School"/>
</node>

... nodes deleted ...

<node id="393585189" visible="true" version="4" changeset="10421071"
     timestamp="2012-01-17T19:38:10Z" user="asciiphil" uid="247807"
     lat="39.2378544" lon="-76.8592826">
     <tag k="highway" v="traffic_signals"/>
</node>

... nodes deleted ...

<node id="4615233124" visible="true" version="1" changeset="45258818"
      timestamp="2017-01-18T00:25:00Z" user="dannykath" uid="2226712"
     lat="39.2348555" lon="-76.8878940"/>

// Here are the nodes for the school zone overlay ...

<node id="111111101" lat="39.209000" lon="-76.94120"/>
<node id="111111102" lat="39.207000" lon="-76.94170"/>
<node id="111111103" lat="39.196800" lon="-76.92377"/>
<node id="111111104" lat="39.195800" lon="-76.92400"/>

<node id="111111105" lat="39.1830"  lon="-76.9700"/>
<node id="111111106" lat="39.2210"  lon="-76.9700"/>
<node id="111111107" lat="39.2220"  lon="-76.9700"/>
<node id="111111108" lat="39.2300"  lon="-76.9700"/>
<node id="111111109" lat="39.2300"  lon="-76.9139"/>
<node id="111111110" lat="39.1935"  lon="-76.9139"/>
<node id="111111111" lat="39.1925"  lon="-76.9139"/>
<node id="111111112" lat="39.1830"  lon="-76.9139"/>
```

```
<way id="145131026" visible="true" version="2" changeset="12667899"
     timestamp="2012-08-09T12:22:19Z" user="asciiphil" uid="247807">
     <nd ref="358249233"/>
     <nd ref="1585804331"/>
     <nd ref="1585804607"/>
     <nd ref="1585804738"/>
     <nd ref="1858630255"/>
     <nd ref="1585804903"/>
     <nd ref="358249233"/>
     <tag k="amenity" v="school"/>
     <tag k="width" v="6"/>
     <tag k="name" v="Clarksville Elementary School"/>
</way>

<way id="145131027" visible="true" version="1" changeset="10361612"
     timestamp="2012-01-11T15:45:44Z" user="asciiphil" uid="247807">
     <nd ref="358249235"/>
     <nd ref="1585803099"/>
     <nd ref="1585803102"/>
     <nd ref="1585803342"/>
     <nd ref="1585803515"/>
     <nd ref="1585803635"/>
     <nd ref="358249235"/>
     <tag k="amenity" v="school"/>
     <tag k="width" v="6"/>
     <tag k="name" v="Clarksville Middle School"/>
</way>

<way id="145131028" visible="true" version="1" changeset="10361612"
     timestamp="2012-01-11T15:45:44Z" user="asciiphil" uid="247807">
     <nd ref="1585803568"/>
     <nd ref="1585803623"/>

     ... nodes for boundary of Pointers Run removed ...

     <nd ref="1585803523"/>
     <nd ref="1585803568"/>
     <tag k="amenity" v="school"/>
     <tag k="width" v="6"/>
     <tag k="name" v="Pointers Run Elementary School"/>
</way>

<way id="145131029" visible="true" version="2" changeset="12667899"
     timestamp="2012-08-09T12:22:20Z" user="asciiphil" uid="247807">
     <nd ref="1585804338"/>
     <nd ref="1585804292"/>

     ... nodes for boundary of River Hill High School removed ...

     <nd ref="1585804338"/>
     <tag k="amenity" v="school"/>
     <tag k="width" v="5"/>
     <tag k="name" v="River Hill High School"/>
```

```
</way>

// Here are the ways for the school zone boundaries ...

<way id="222222221">
 <nd ref="111111102"/>
 <nd ref="111111104"/>
 <nd ref="111111111"/>
 <nd ref="111111112"/>
 <nd ref="111111105"/>
 <nd ref="111111106"/>
 <nd ref="111111102"/>
 <tag k="name" v="Pointers Run Elementary School"/>
 <tag k="amenity" v="schoolzone"/>
 <tag k="width" v="8"/>
</way>

<way id="222222222">
 <nd ref="111111105"/>
 <nd ref="111111108"/>
 <nd ref="111111109"/>
 <nd ref="111111112"/>
 <nd ref="111111105"/>
 <tag k="name" v="Clarksville Middle School"/>
 <tag k="amenity" v="schoolzone"/>
 <tag k="width" v="2"/>
</way>

<way id="222222223">
 <nd ref="111111105"/>
 <nd ref="111111108"/>
 <nd ref="111111109"/>
 <nd ref="111111112"/>
 <nd ref="111111105"/>
 <tag k="name" v="River Hill High School"/>
 <tag k="amenity" v="schoolzone"/>
 <tag k="width" v="2"/>
</way>

<way id="222222224">
 <nd ref="111111101"/>
 <nd ref="111111107"/>
 <nd ref="111111108"/>
 <nd ref="111111109"/>
 <nd ref="111111110"/>
 <nd ref="111111103"/>
 <nd ref="111111101"/>
 <tag k="name" v="Clarksville Elementary School"/>
 <tag k="amenity" v="schoolzone"/>
 <tag k="width" v="8"/>
</way>

<relation id="936304" visible="true" version="17" changeset="43751383"
          timestamp="2016-11-18T01:50:43Z" user="nyuriks" uid="339581">
```

```
     <member type="way" ref="92449187" role="outer"/>
     <member type="way" ref="86310899" role="outer"/>

     ... ways for Howard County Boundary removed ...

     <member type="way" ref="394717244" role="outer"/>
     <member type="way" ref="394717252" role="outer"/>
     <tag k="admin_level" v="6"/>
     <tag k="attribution" v="USGS 2001 County Boundary"/>
     <tag k="border_type" v="county"/>
     <tag k="boundary" v="administrative"/>
     <tag k="name" v="Howard County"/>
     <tag k="type" v="boundary"/>
</relation>

<relation id="961626" visible="true" version="30" changeset="43016223"
     timestamp="2016-10-19T19:00:10Z" user="ElliottPlack" uid="105946">
     <member type="way" ref="5986058" role=""/>
     <member type="way" ref="130467897" role=""/>
     <member type="way" ref="5973668" role=""/>

     ... ways for MD Route 108 removed ...

     <member type="way" ref="5269975" role="forward"/>
     <member type="way" ref="137363577" role=""/>
     <tag k="is_in" v="US:MD"/>
     <tag k="network" v="US:MD"/>
     <tag k="ref" v="108"/>
     <tag k="route" v="road"/>
     <tag k="type" v="route"/>
</relation>

<relation id="1354489" visible="true" version="54" changeset="43751383"
     timestamp="2016-11-18T01:51:30Z" user="nyuriks" uid="339581">
     <member type="way" ref="93131606" role=""/>
     <member type="way" ref="110779250" role=""/>

     ... ways for Maryland Route 32 removed ...

     <member type="way" ref="31363871" role="west"/>
     <member type="way" ref="107999420" role="west"/>
     <tag k="is_in" v="US:MD"/>
     <tag k="network" v="US:MD"/>
     <tag k="ref" v="32"/>
     <tag k="route" v="road"/>
     <tag k="type" v="route"/>
</relation>

<relation id="1964719" visible="true" version="1" changeset="10361612"
          timestamp="2012-01-11T15:46:06Z" user="asciiphil" uid="247807">
          <member type="way" ref="145131126" role="outer"/>
          <member type="way" ref="145131088" role="inner"/>
          <tag k="building" v="school"/>
          <tag k="type" v="multipolygon"/>
```

```
 </relation>

 <relation id="1964720" visible="true" version="1" changeset="10361612"
           timestamp="2012-01-11T15:46:06Z" user="asciiphil" uid="247807">
           <member type="way" ref="145131127" role="outer"/>
           <member type="way" ref="145131118" role="inner"/>
           <tag k="building" v="school"/>
           <tag k="type" v="multipolygon"/>
 </relation>

 <relation id="1964721" visible="true" version="1" changeset="10361612"
           timestamp="2012-01-11T15:46:07Z" user="asciiphil" uid="247807">
           <member type="way" ref="145131104" role="outer"/>
           <member type="way" ref="145131128" role="inner"/>
           <member type="way" ref="145131112" role="inner"/>
           <tag k="building" v="school"/>
           <tag k="type" v="multipolygon"/>
</relation>
</osm>
```

Bibliography

[1] Apache Jena:. An Open Source Java framework for building Semantic Web and Linked Data Applications. For details, see https://jena.apache.org/, 2016.

[2] Austin M.A., Baras J.S., and Kositsyna N.I. Combined Research and Curriculum Development in Information-Centric Systems Engineering. In *Proceedings of the Twelth Annual International Symposium of The International Council on Systems Engineering (INCOSE)*, Las Vegas, USA, July 2003.

[3] Austin M.A., Delgoshaei P. and Nguyen, A. Distributed System Behavior Modeling with Ontologies, Rules, and Message Passing Mechanisms. *Procedia Computer Science*, 44:373 – 382, 2015. 2015 Conference on Systems Engineering Research.

[4] Austin M.A., Mayank V., and Shmunis N. PaladinRM: Graph-Based Visualization of Requirements Organized for Team-Based Design. *Systems Engineering: The Journal of the International Council on Systems Engineering*, 9(2):129–145, May 2006.

[5] Batty M. *Cities and Complexity: Understanding Cities with Cellular Automata, Agent-based Models and Fractals*. The MIT Press, London, England, 2005.

[6] Batty M. *The New Science of Cities*. The MIT Press, Cambridge, MA, 2013.

[7] Berners-Lee T., Hendler J., and Lassila O. The Semantic Web. *Scientific American*, pages 35–43, May 2001.

[8] Bonino D. and Corno F. *DogOnt - Ontology Modeling for Intelligent Domotic Environments*, pages 790–803. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[9] Delgoshaei, P. and Austin, M.A. and Pertzborn, A. A Semantic Framework for Modeling and Simulation of Cyber-Physical Systems. *International Journal On Advances in Systems and Measurements*, 7(3-4):223–238, December 2014.

[10] Delgoshaei, P. and Austin, M.A and Veronica, D.A. A Semantic Platform Infrastructure for Requirements Traceability and System Assessment. *The Ninth International Conference on Systems (ICONS 2014)*, February 2014.

[11] Falquet G., Metral C., Teller J., and Tweed C. *Ontologies in Urban Development Projects*. Springer, 2005.

[12] Feigenbaum L., 2006. Semantic Web Technologies in the Enterprise.

[13] Fridenthal S., Moore A., and Steiner R. *A Practical Guide to SysML*. MK-OMG, 2008.

[14] Geroimenko V., and Chen C. (Eds). *Visualizing the Semantic Web: XML-based Internet and Information Visualization*. Springer, 2003.

[15] Google Maps (2016), 8115 Baltimore Ave. College Park, MD.

[16] Hendler J. *Agents and the Semantic Web. IEEE Intelligent Systems*, pages 30–37, March/April 2001. Available on April 4, 2002 from http://www.computer.org/intelligent.

[17] Hohpe G. and Woolf B. *Enterprise Integration Patterns: Designing, Building and Deploying Message Passing Solutions*. Addison Wesley, 2004.

[18] Horrocks I. *Ontologies and the Semantic Web*. Communications of the ACM, 51(12):58-67, December, 2008.

[19] Horrocks I., Patel-Schneider P. F., and Van Harmelen F. *From SHIQ and RDF to OWL: The Making of a Web Ontology Language*. Journal of Web Semantics, 1(1):7-26, 2003.

[20] Ibsen C., Antsey J., and Hadrian Z. *Camel in Action*. Manning Publications Company, 2010.

[21] Jackson D. *Dependable Software by Design. Scientific American*, 294(6), June 2006.

[22] Lee E.A., 2003. Model-Driven Development – From Object-Oriented Design to Actor-Oriented Design, Presentation at Workshop for Software Engineering for Embedded Systems, From Requirements to Implementation, Chicago,.

[23] Mahmoud Q.H. Getting Started With the Java Rule Engine API (JSR 94): Toward Rule-Based Applications. *Sun Microsystems*, 2005. For more information, see http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html (Accessed, March 10, 2008).

[24] Maier, M.W., and Rechtin E. *The Art of Systems Architecting, 2nd Edition*. CRC Press, London, 2000.

[25] Mosteller M., Austin M.A., Yang S., and Ghodssi R. *Platforms for Engineering Experimental Biomedical Systems. IEEE Systems Journal*, PP(9):1–11, September 2014.

[26] Muller D. Requirements Engineering Knowledge Management based on STEP AP233. 2003.

[27] Myers C.A., Slack T., and Singelmann J. Social Vulnerability and Migration in the Wake of Disaster: The case of Hurricanes Katrina and Rita. *Population and Environment*, 29:271–291, 2008.

[28] Open Street Map (OSM). https://www.openstreetmap.org (Accessed April 3, 2017). 2017.

[29] OptaPlanner (2016), A Constraint-Satisfaction Solver. For details, see: https://www.optaplanner.org (Accessed, Jan 4., 2017).

[30] OWL:. The Web Ontology Language, See http://www.w3.org/TR/owl-features/ (Accessed, February 2017).

[31] Rudolf G. Some Guidelines For Deciding Whether To Use A Rules Engine. 2003. Sandia National Labs.

[32] Segaran T., Taylor J., Evans C. *Programming the Semantic Web.* O'Reilly, Beijing, 2009.

[33] Selberg S. and Austin M.A. Toward an Evolutionary System of Systems Architecture. In *18th Annual International Symposium of The International Council on Systems Engineering (INCOSE 2008)*, Utrecht, The Netherlands, June 15-19 2008.

[34] Stelting S. and Maassen O. *Applied Java Patterns.* SUN Microsystems Press, Prentice-Hall, 2002.

[35] Tidwell D. *XSLT.* O'Reilly and Associates, Sebastopol, California, 2001.

[36] White House (2003), The National Strategy for the Physical Protection of Critical Infrastructures and Key Assets. Washington, DC.

[37] White R. Cities and Cellular Automata. *Discrete Dynamics in Nature and Society*, 2:111–125, 1998.

[38] Whiting E. Geometric, Topological and Semantic Analysis of Multi-Building Floor Plan Data. *MS Thesis, Master of Science in Architecture Studies, MIT*, June 2006.

[39] Willmott A. Glassbox Game Engine (Simulation engine powering Simcity 2013). *Game Developers Conference, San Francisco, CA*, 2012.

[40] XML Stylesheet Transformation Language (XSLT). See http://www.w3.org/Style/XSL. 2002.