Architecture for Ontology-supported Multi-context Reasoning Systems

Andrew LeClair[a,*], Jason Jaskolka[b], Wendy MacCaull[a,c], Ridha Khedri[a]

[a]*Department of Computing and Software, McMaster University*
1280 Main Street West, Hamilton ON L8S 4L7, Canada
[b]*Department of Systems and Computer Engineering, Carleton University*
1125 Colonel By Drive, Ottawa ON K1S 5B6, Canada
[c]*Department of Computer Science, St. Francis Xavier University*
PO Box 5000, Antigonish NS B2G 2W5, Canada

Abstract

Modern smart systems such as those needed for Industry 4.0 integrate data from various sources and increasingly require that data be contextualized with domain knowledge. The integration and contextualization of data allows for the advanced reasoning needed to generate knowledge grounded in the data under consideration. In this paper, we propose an architecture for an ontology-supported multi-context reasoning system which inherently supports a number of desired system qualities including data transparency, system interactivity, and graceful aging. The architecture is inspired by the Presentation-Abstraction-Control architecture style, which is an interaction-based architecture. Our architecture uses a two level hierarchy with three agents and can incorporate and utilize multiple contexts. It is flexible, supporting an interface between data and users, highly interactive, and easily maintained. The evolution of data is isolated to a single component of the system and therefore does not cascade to several others. A domain of application can be easily determined by the use of archetypes and domain-specification components. Our architecture is demonstrated using a case study involving data from the city of San Francisco.

Keywords: software architecture, knowledge-based system, ontology-based system, intelligent system

1. Introduction

In the architecture of smart systems, decentralization of data stores and agents brings the desired properties such as dependability, modifiability, and extendability of new agents. For example, Industry 4.0, the so called fourth industrial revolution, speaks of having multiple co-operating agents communicate via wireless technologies, making decisions based on data from distributed sensors or other data sources, to perform duties respective to the industry [1]. The decentralization of smart systems has called for new solutions for the problems related to having numerous smart agents. In particular, with the numerous agents in an ecosystem, it is required that its agents be intelligent, i.e., they can make their own decisions, and are able to share their knowledge with each other, allowing for a system where agents empower each other. The inclusion of knowledge in a smart system is a complex topic which requires an understanding of what type of knowledge an agent should know [2].

Ontologies are used to conceptualize a particular domain of discourse to facilitate the communication among agents, allowing for the ability to reason and acquire new knowledge [3]. An ontology is defined as a "*formal, explicit specification of a shared conceptualization*" [4], and is often thought of as a relational structure. The domain of discourse is conceptualized using a set of concepts, a set of relations which relate these concepts to one another, and a set of axioms formalized using some specified language. By conceptualizing a domain with an ontology, the agents can communicate to one another using the language of the ontology, and can make decisions using the domain knowledge of the ontology in conjunction with their data. Ontologies have been incorporated in several Industry 4.0 applications [1, 5], as well as in domains such as smart homes [6].

Some domains, such as healthcare, involve situations where each of the agents of the system need to have its own view of the domain [7]. In this case, the agents are interfacing with human actors, such as a doctor, nurse, and pharmacist, who interact with the healthcare system. Each has their own unique interaction with the system. For example, the nurse requires the system to aid in their treatment of the patient, whereas the doctor requires the system to aid in the diagnosis of the patient, and the pharmacist requires the system to aid in prescribing the treatment. For such a system, the domain must be contextualized according to the acting agent. Thus, the system must present any information that can be used to characterize the specificity of domain concepts [8] and express the specializations of the shared model (i.e., ontology) that define a given local model [9]. The design of systems that incorporate ontologies, such as O414 and ORArch in [1], do not account for providing the multiple contexts.

We envision an ontology-supported multi-context reasoning (OMR) system as the decision-making component of a larger hardware and software ecosystem with the ability to support

*Corresponding author
Email addresses: leclaial@mcmaster.ca (Andrew LeClair), jason.jaskolka@carleton.ca (Jason Jaskolka), wmaccaul@stfx.ca (Wendy MacCaull), khedri@mcmaster.ca (Ridha Khedri)

multiple contexts. Such an OMR system is applicable in a domain with the following characteristics: the domain requires several participating agents each having their own view of the domain, multiple data sources are used to populate the domain, and a conceptualization of the domain is required for the agent's reasoning. The Architecture for Ontology-supported Multi-context Reasoning (AOMR) systems is appropriately designed with respect to the design principles associated with the mentioned smart systems and Industry 4.0: interoperability, virtualization, decentralization, real-time capability, service orientation, and modularity. We foresee the potential to use OMR systems in numerous emerging technological and industrial paradigms such as ambient intelligence [10], adaptive e-learning [11], or knowledge management [12].

AOMR is based on a variation of the popular presentation-abstraction-control (PAC) architecture style. We argue that the PAC architecture style can accommodate the demands of OMR systems without any change to the system: the design is open for addition and closed for modification. To demonstrate these qualities, we provide a design scenario for an illustrative OMR system. The example system uses an ontology and open data sets for the City of San Francisco. The case study of the City of San Francisco was chosen because of the ease of communicating multiple contexts for a city, and simplicity in communicating the highlights of AOMR. We show that the proposed architecture supports the design of an OMR system that provides the flexibility required of modern smart system applications.

The rest of this paper is organized as follows. Section 2 introduces AOMR and its components within the PAC architecture style. Section 3 further elaborates the components of AOMR using a design scenario for an OMR system. Section 4 presents a prototype implementation of an OMR system following AOMR, motivated by the City of San Francisco domain. Section 5 discusses AOMR, and how it meets the goals of an OMR system. Section 6 positions our contributions in the existing literature. Finally, Section 7 concludes the paper and highlights the directions of our future work.

2. The Architecture of Ontology-supported Multi-view Reasoning Systems

We introduce the architecture, AOMR, that provides a systematic way to organize the concepts and their relationships such that multiple contexts are captured and represented as needed. We use the term *context* to describe a representation of a domain in which the data is to be interpreted. More specifically, we consider a context model made of two distinct parts. The first is the *domain context* which specializes the archetypal concepts, which are generic concepts, and relationships of the shared model with concrete types and additional properties. The second is the *data context* which associates data instances to the specific concepts defined in the context model. Therefore, when we say *context model*, we mean the combination of both the domain context(s) and data context(s). In [13], we find a wide discussion on the notion of context and its many definitions. However, the authors of [13] conclude that, in general, a context is the set of information that can be used to characterize the situation of an entity/agent. This general definition is in line with our understanding of context. Indeed, the data from the data part of the context model (i.e., data context) when interpreted within the domain given by the second part of the context model (i.e., domain context) gives us a set of information that characterizes the situation of the agent or entity that uses the context model.

In OMR systems, concepts under consideration need to be processed using different domain contexts, each reflecting the concepts required to represent a particular domain. For instance, a concept *Park* can be contextualized in several domains. In a geographical domain, a *Park* is a *Location* which is a set of co-ordinates that give the park boundaries. The *Park* concept can be also contextualized in an ecological domain where a *Park* is a habitat in which a set of organisms live. In the ontology of this domain, we would find the *Park* concept related to *Species* concept. The concept *Park* has two different domain contexts: one for the geographical domain, and one for the ecological domain. The reasoning in this case is multi-context as it is supported by two domain contexts. If the reasoning component of the system is capable of accommodating the reasoning within several domain contexts, then we can enhance the capabilities of the system in generating knowledge to support informed decisions. These domain contexts should be added with minimal modification to the system. For instance, we should be able to add a new domain context in which we understand the concept *Park* as a botanical concept giving us information about the plants thriving within the location of the park.

Although an OMR system uses an ontology to conceptualize the domain, the conceptualization of the ontology should be independent of the particular modelling or implementation language. This approach is similar to how the design choices, when engineering a software system, are made independent of the implementation. We introduce the goals of the architecture which are motivated by the demands of smart systems such as the conceived ones for Industry 4.0 and Health 4.0.

2.1. Goals of Our Architecture

In what follows, consider three actors who are stakeholders involved in the development of an OMR system: the *customer*, the *software engineer*, and the *user*. The customer is the actor who requests the OMR system. The software engineer is the actor who designs and implements the system to meet the customer's requirements (listed as competency questions). Finally, the user is the actor that will be interacting with the system. The following are the three primary goals for an OMR system:

Goal 1: Data Transparency

Data transparency is defined in [14] as "*the ability of a subject to effectively gain access to all information related to data used in processes and decisions that affect the subject*" and is mentioned in fields that utilize multiple data sets, where the data may be in various locations, and in various formats. Examples of fields that highlight the need for data transparency are Artificial Intelligence, Machine Learning, and Industry 4.0. Data heterogeneity is another term used to describe when data has vari-

ous formats, but unlike data transparency, it does not include the situation where there are various locations of data [15]. Since it should not be assumed that the data provided to the software engineer is from the same location, same format, or a same source, it is crucial that the architecture satisfies the property of data transparency. It should easily interface with the data, regardless of data location or format.

Goal 2: System Interactivity

In an Industry 4.0 setting, it is essential to handle numerous service requests in real-time. In an OMR system, the service requests are the queries which are associated to the user's context. A user should be able to interact with the system designed and developed by the software engineer to choose the context in which they wish to view the domain as well as their query and, possibly, the reasoner. Thus, to provide a good user experience, the architecture must promote interactivity.

Goal 3: Graceful Aging of the System

In a domain of application where new system components (such as data sources) are constantly added or removed, or existing connections are modified, the architecture should be easily maintainable, modifiable, and extensible. Maintainability is a primary concern with regards to the evolution of the domain. It is expected that as new data is collected, or new sources of data are added, the understanding of concepts or parts of the domain will change. These changes must be localized to their respective components (i.e., changes in the data effecting only the components related to the data). Additionally, the Industry 4.0 setting demands the creation of systems that are modular. It should be simple to connect the numerous participating software components, and it should be simple to update the system as new products or system parts are introduced.

2.2. Overview of AOMR

As we introduced, AOMR follows a variation of the PAC architecture [16], which is characterized as a system that is composed of multiple agents, where the agents are organized as a hierarchy. The agents in a PAC system are not the same as the agents of a smart system as described in the introduction. A smart system agent interacts with the system via queries, where as PAC agent is a distinguished part of the system itself composed of three components–a presentation, an abstraction, and a control–which together form a triple. It is similar to the Model-View-Controller (MVC) architecture in that it is an interaction-oriented architecture, but unlike MVC, which is a single triple, a PAC system is composed of a hierarchy of triples, i.e., agents. PAC is a hierarchy of MVC instances with the condition that the presentation component and abstraction component communicate only through the controller component. AOMR is composed of a two level hierarchy with three agents, as shown in Figure 1. The lower-level of the hierarchy has two agents: the interpretations agent, and the data agent. The interpretations agent is responsible for the conceptualization of a domain. By first conceptualizing a domain with generic concepts, the software engineer can capture the domain at a glance. These generic concepts are referred to as *archetypes*. By remaining generic, the archetypes are reusable across domains as they are not tied to any single domain. Thus, a software engineer may be able to reuse an archetype from a prior project. A *domain-specific concept* is created by contextualizing an archetype. The contextualization of an archetype is the process of enriching it with further attributes or specifying how an attribute is to be understood (i.e., a typing). This process results in a domain context of the archetype: the domain-specific concept. By inspecting the archetypes and domain-specific concepts, the customer can determine if the domains have been sufficiently captured. On the other hand, the data agent is responsible for creating and storing the mappings between the domain-specific concepts and data. The process of mapping a specific set of datasets to a domain-specific concept results in a data contextualization for that domain-specific concept. In other words, the data agent saves the data contexts of the domain-specific concepts. To avoid ambiguity or confusion of the usage for contextualization, we refer to the process of contextualizing an archetype as *interpreting*. Thus, the interpretations agent is responsible for knowing and applying the interpretations (i.e., the domain contexts). Finally, the knowledge agent is responsible for coordinating the system and the knowledge therein. It is aware of the context models, so when the user submits a query, it assembles the context model and necessary domain-specification component in a format that can be reasoned on to answer the query.

As it is a PAC architecture, each agent is composed of a presentation, an abstraction, and a control component. Rather than taking the sensorial (visual—such as screen layout—or other) representation for the *Presentation* component, we take the ontological presentation of the world (i.e., set of domains) in which the data given by the *Abstraction* component is to be interpreted. Among these ontological presentations, we consider both the archetypal concepts used within a set of domains and domain-specific concepts to support the reasoning tasks of OMR systems. Each type of component is denoted by a color: white for presentation, grey for abstraction, and black for control. The two levels of the hierarchy are the lower *Theory Level* and the upper *Knowledge Level*. With respect to Figure 1, the *Archetypes Component* provides the shared model, and the *Interpretations Coordinator* and the *Data Coordinator* together construct the context model as described in [9].

2.3. The Theory Level of AOMR

The *Theory Level* is the lower-level of the hierarchy and is comprised of two agents: the interpretations agent and the data agent. Together, these two agents enable creation of generic conceptualizations with archetypes that can be contextualized to capture specific domains.

2.3.1. Interpretations Agent

The interpretations agent is responsible for scoping out the domain by establishing the domain contexts. It is composed of the following components: the archetypes component, the interpretations coordinator, and the domain-specification components.

The first component is the archetypes component, and it is responsible for storing the archetypes used in the system.

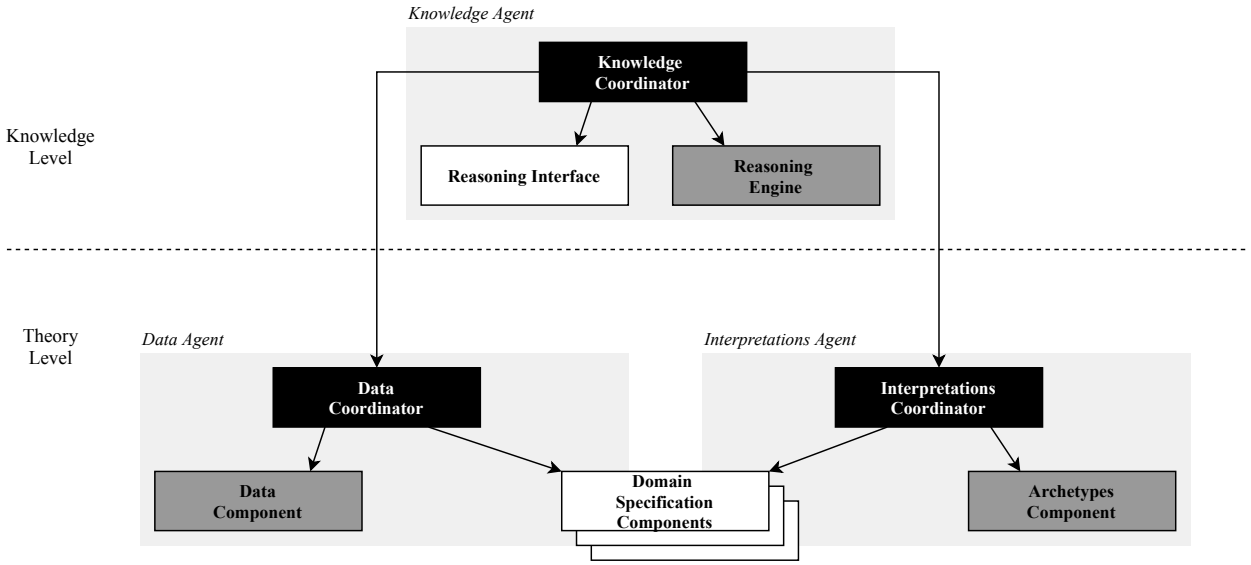


Figure 1: AOMR demonstrating the PAC architecture style.

The archetypes are the generic, domain-independent concepts that are interpreted to conceptualize the domain. As such, the archetypes form the *conceptual skeleton* of the domain.

Archetypes must be both reusable and easily accessible. This is achieved by having the archetypes component be modular. The archetypes component is an abstraction component from the PAC style; an archetypes component from a different domain can be plugged into an interpretations coordinator and made to function with minimal changes to the interpretations coordinator. The ability to use archetypes components from another system allows the software engineer to create domain-specific concepts from archetypes of another design exercise when possible.

The second component is the interpretations coordinator, and it is the control component of the interpretations agent. The interpretations agent is responsible for two tasks: creating domain-specific concepts, and hiding details of the domain conceptualization from the rest of the system. When the knowledge coordinator begins to assemble the ontology in response to a query, it will require the necessary domain-specification component from the interpretations coordinator. Any details regarding a domain-specification component or the domain-specific concepts therein, such as the archetypes that are interpreted, is hidden from the knowledge coordinator.

The final component is the set of domain-specification components, each component is the description of how an archetype is interpreted to capture the details and relationships of the domain. As such, it provides the domain-specific knowledge of the domain. It is a specific and unique way of understanding the archetypes. As the domain contexts that are required are outlined via the requirements, the domain-specification components that will be created are themselves determined via the requirements.

A single domain-specification component is made of one or many domain-specific concepts. These concepts are defined in one of two ways: by the interpretation of an archetype, or axiomatically. Defining a domain-specific concept axiomatically allows the domain-specific concept to be defined as an intermix of other domain-specific concepts.

AOMR follows a variation of the PAC architecture wherein the domain-specification components are shared between the interpretations agent and the data agent. Whereas the interpretations coordinator establishes how the domain-specification component is an interpretation of archetypes, the data coordinator establishes how the domain-specification component relates to the data of the domain. The relationship between the data coordinator and the domain-specification component is further discussed in Section 2.3.2.

2.3.2. Data Agent

The data agent is responsible for establishing the data contexts of the domain. By defining data contexts for the domain-specification components, which will have domain contexts defined by the interpretations agent, we create a full context model for the domain-specification component. The data agent is composed of the following components: the data component, the data coordinator, and the domain-specification components.

The first component is the data component which manages the data or information provided by the customer by interfacing to the data sources. Since the data may be off-site or in multiple locations, it is essential that the data component is able to retrieve the data, regardless of location or format, needed for the data coordinator. In this way, the data component acts as a gateway through which the data can flow. Together, the data gateway and the data sources perform the functionality of handling requests related to the data, such as providing the data to create a data context. Because all data requests are accomplished by the data component, it is considered an abstraction component from the PAC architecture. The idea with the data component is to have the ability to "plug-in" new data source connections as they become available or as they are needed. Often, data sources will provide a template that inform the data component

of the format of the data so that it may be used for its particular purpose and in its particular application domain.

The second component is the data coordinator, which is the control component for the data agent. Whereas the interpretations coordinator is responsible for making domain contexts by interpreting an archetype into a domain-specific concept, the data coordinator is responsible for making data contexts by associating a specified attribute with a subset of data (such as a column from a table). A data context may be in any format or data structure that provides a link between the domain-specific concepts within the domain-specification component and the data it maps to. For example, a data context can be made using a hashmap.

The final component is comprised of the domain-specification components, shared with the Interpretations Agent and introduced in Section 2.3.1.

2.4. The Knowledge Level of AOMR

The *Knowledge Level* is the upper-level of the hierarchy, and is composed of a single agent: the knowledge agent. While the *Theory Level* is responsible for any knowledge representation concerns, the *Knowledge Level* provides the reasoning capabilities necessary for the system to be a reasoning system. The knowledge agent is comprised of the reasoning interface, the reasoning engine, and the knowledge coordinator.

The reasoning interface is the presentation that allows the user to query the system. Since the *Knowledge Level* is separated from the *Theory Level*, the reasoning interface allows the user to choose a context model without being exposed to the archetypes or datasets. The experience of the user is focused entirely on submitting a query to the system that is answered based on their selected context model and reasoner.

The reasoning engine is the abstraction component of the knowledge agent and manages the reasoners that can be used to answer a submitted query. Much like how the data component interfaces with off-site data sources, the reasoning engine interfaces with any plugged-in reasoners. Once connected, the reasoner can be used for answering queries, so long as any requirements to use the reasoner are fulfilled. For instance, for many DL-based reasoners, the ontology must be written in W3C Web Ontology Language (OWL) (e.g., [17, 18, 19]). To achieve this, the software engineer should provide both the reasoner as well as any requirements (such as necessary ontology representation) to the reasoning engine. When the knowledge coordinator contacts the reasoning engine, it is then aware of any requirements the ontology must satisfy if a specific reasoner is to be used. There can be many different reasoners, or different types of reasoners, each having their own interface with the reasoning engine.

The knowledge coordinator, as the control of the knowledge agent, is the control of the entire system. Given a query, context model, and reasoner, the knowledge coordinator must submit the query to the reasoner with the given context of the domain. The context of the domain is determined by retrieving the context model and domain-specification component from the *Theory Level* and possibly transforming it into the format required by the reasoning engine (such as OWL). Once the reasoning process has been completed, the results are returned to the reasoning interface.

3. Example System Design Scenario

In this section, we design a simple OMR system according to AOMR. The data that is used to populate the domains comes from the open data sets for the city of San Francisco [20]. We have chosen this dataset over a dataset from an Industry 4.0 setting because of its public-access, size and the heterogeneity of the data, and so that the reader may understand the case study without any manufacturing domain expertise. Although the data does not specifically relate to manufacturing lines or healthcare, it is intended to illustrate how multiple domains can be represented. In this example, the world is composed of three domains: a real estate domain, a city infrastructure domain, and a tourism domain. The customer expresses their concerns about the world through competency questions. The concerns are related to domains, as follows:

Real Estate Domain

1. Which neighborhoods have 2-bedroom houses?
2. Tell me about the houses on Castro St.
3. Which houses are on a street with a tall building and a Muni stop?

City Infrastructure Domain

1. Which streets require repaving?
2. Where are the parks that need to be maintained?
3. Which areas of the city need to be inspected?

Tourism Domain

1. What is a route for an *Ant-Man*[1] tour?
2. Where are famous San Francisco sites?
3. Which parks were featured in a film?

From these competency questions, it is apparent how the different domains require the same data, but use it in different ways. For instance, all three domains use the concept of *street*, and thus, all use data about the streets of San Francisco. However, the understanding of a street differs based on the domain. In the real estate domain, a street is an entity on which buildings or amenities (e.g., Muni stop) exist. In the city infrastructure domain, a street is an entity which has a pavement score index (PCI) and speedlimit, and requires maintenance. In the tourism domain, a street is a filming location. However, all three domains agree on the constituent data for the streets. Afterall, a street, such as *Castro St.*, is a street no matter which domain it is in. With these questions as the basis for illustration, we describe the realization of each component of AOMR. This example is purposely designed to highlight particular aspects and features of AOMR.

[1] Ant-Man is a 2015 Marvel Comics film that featured several San Francisco locations.

Table 1: Archetypes and their attributes for inclusion in the *San Francisco Ontology*.

Archetype	Attributes
Area	*name* *location*
Route	*name* *path*
Building	*name* *address*

3.1. Realizing the Theory Level of AOMR

As it was introduced in Section 2.3, the *Theory Level* of AOMR is responsible for conceptualizing a domain through two agents: the interpretations agent and the data agent. The process of conceptualizing a domain is multi-step. First, the archetypes must be defined. Second, the domain-specification components that are used to answer the competency questions must be determined. Finally, the data contexts which contextualize the domain-specific concepts with data to answer the queries of the three domains must be created. In effect, this realization provides the ontology for the OMR.

3.1.1. The Archetypes

The first step in conceptualizing a domain is to determine the archetypes that will be used to answer the competency questions. For this example, all archetypes are created from scratch, but in most situations, archetypes may be reused from a system previously designed.

From our example, it can be seen that the real estate domain and city infrastructure domain both require the conceptualization of a location: the real estate domain is asking for the location of Muni stops, and the city infrastructure domain is asking for areas that need inspection. Therefore, we require some concept which has an attribute of *location*. The exact understanding of a location is not yet prescribed so as to not limit a specific domain. If we were to limit a location to be a set of GPS co-ordinates, then we may prevent the real estate domain from querying the location of Muni stops by their street or intersection. Thus, an archetype is created with this attribute of *location* and titled *Area*. In Table 1, we present the archetypes for our *San Francisco Ontology* required to answer our competency questions. They include the archetype *Area* described above, and two additional archetypes: *Route* and *Building*.

Similar to *Area*, *Route* and *Building* are created as a result of the need for specific attributes that may cross many domains. A *Route* captures the notion of a path, such as the one described in the tourism domain. Although this property of route is not immediately present in other domains, it is defined as an archetype so that it can be reused in another domain if the need arises. The third archetype of *Building* conceptualizes the physical entity that exists at an *address*.

The archetype is a typical unifying and abstract concept that can be instantiated and/or extended across the domains (i.e., to be used in varying context models). As such, an archetype needs to be provided at an appropriate granularity; we could have defined a single archetype of *Place* which would encompass both an *Area* and *Building*, but that would abstract away what makes those archetypes significant. A *Building* is intended to be a physical entity with an address whereas an *Area* is any conceivable boundary at a location. This distinction allows us to understand a Muni stop as some *Area* because it certainly has a location, but does not have an address. Thus, if we are *too* general in defining an archetype *Place*, we would lose this distinction between *Area* and *Building*, and gain no significant conceptualization of the domain. It might also be the case that archetypes unnecessary for any competency questions are defined. For instance, a *Person* archetype which has attributes such as a name and age, would never be interpreted as no question uses any notion of a person.

[Real Estate Domain]

Street **instantiates archetype** *Route* **with**
name : String;
path : Set⟨Integer⟩;

House **instantiates archetype** *Building* **with**
name : String;
address : Street;
neighborhood : String;
listing : String;

Muni Stop **instantiates archetype** *Area* **with**
name : String;
location : Co-ordinates;
onStreet : Street;
atStreet : Street;
shelter : Boolean;

Tall Building **instantiates archetype** *Building* **with**
name : String;
address : Street;
height : Float;
constructionYear : Int;

Busy House **refines concept** *House* **with**
house : House;
$\exists(m \mid m \in MuniStop : house.address = m.onStreet \lor house.address = m.atStreet)$;
$\exists(t \mid t \in TallBuilding : house.address = t.address)$;

Figure 2: Domain-specification component of the real estate domain.

3.1.2. The Domain-Specification Components

The domain-specification components are a collection of domain-specific concepts. The domain-specific concepts are themselves created either by interpreting archetypes or by using axioms. A domain is captured by a single domain-specification component. For our example, the three domains and their respective domain-specification components are given in Figures 2, 3, and 4, respectively. These domain-specification components are written in Z-like notation.

In Figure 2, the archetype *Route* has been interpreted as a domain-specific concept called *Street*. Similarly, we see the archetype *Building* interpreted as a *House*. *House* specifies the types of the two attributes *name* and *address* from the archetype *Building* as *String* and *Street*, respectively. The typing of street is itself a domain-specific concept that belongs to the domain-specification component. In addition to this,

[City Infrastructure Domain]

Maintainable Area **instantiates archetype** *Area* **with**
name : String;
location : Co-ordinates;

Maintainable Street **refines concept** *Area* **with**
name : String;
location : Co-ordinates;
PCI : Int;
speedLimit : Int;
pedestrianVolume : Int;

Park **refines concept** *Area* **with**
name : String;
location : Co-ordinates;
squareFeet : Float;
parkScore : Float;

Dire Street **refines concept** *Maintainable Street* **with**
street : MaintainableStreet;
$street.speedLimit \geq 35$;
$street.PCI \leq 55$;

Small Dire Park **refines concept** *Park* **with**
park : Park;
$park.parkScore \geq 80$;
$park.squareFeet \leq 100,000$;

Dire Area **refines concept** *Maintainable Area* **with**
area : Maintainable Area;
Dire Street $\cup$ Small Dire Park;

Figure 3: Domain-specification component of the city infrastructure domain.

[Tourism Domain]

Film Site **instantiates archetype** *Area* **with**
name : String;
location : filmLocation;
yearFilmed : Int;
hasFilm : String;

Film Location **instantiates archetype** *Area* **with**
name : String;
location : String;

Tour **instantiates archetype** *Route* **with**
name : String;
sights : Set⟨Film Site⟩;

Street Film Location **refines concept** *Film Location* **with**
filmSite : Film Site;
$\exists(s \mid s \in Street : filmSite.location = s.name)$;

Park Film Location **refines concept** *Film Location* **with**
filmSite : Film Site;
$\exists(p \mid p \in Park : filmSite.location = p.name)$;

Ant-Man Tour **refines concept** *Tour* **with**
tour : *Tour*;
$\forall(s \mid s \in tour.sights : s.hasFilm :=$ 'Ant-Man');

Figure 4: Domain-specification component of the tourism domain.

the attributes *neighborhood* and *listing* (and their typings) are added to the domain-specific concept to allow for answering competency questions, such as "*Which neighborhoods have 2-bedroom houses?*".

In Figure 2, all domain-specific concepts are defined via the interpretation of an archetype, except for *Busy House*, which is defined with a first-order logic axiom. It states that a *Busy House* is a *House* which has both a Muni stop and tall building nearby, where a Muni stop and tall building are both domain-specific concepts. Instead of being the result of an intepretation of an archetype, *Busy House* is instead a further restriction of the already existing domain-specific concept *House*.

The remaining tables depict the domain-specification components for the remaining domains. Figure 3 shows the domain-specification component for the city infrastructure domain and it contains the domain-specific concepts of a *Maintainable Area*, *Maintainable Street*, *Dire Street*, *Park*, *Small Dire Park*, and *Dire Area*. The domain-specific concepts are necessary to define what a park is, or what a street needing maintenance is. Figure 4 shows the domain-specification component for the tourism domain, and contains the domain-specific concepts that describe what a *Film Site* or *Tour* is.

Each of the domain-specification components shown in Figures 2, 3, and 4 can be viewed as different ontologies (representative of the TBox). As we discuss in Section 3.1.4, each of these ontologies are then instantiated by the data component (representative of the ABox). This instantiation is also shown later in Figure 5. Further, because of the separation of concerns provided by the *Theory Level* of AOMR, one of the benefits of adopting the architecture is that we can consider each of the three contexts for the example OMR in an organized way, rather than as a single monolithic ontology.

3.1.3. The Data

In a typical scenario, the customer provides the software engineer with the data that populates the domains. We hand-selected the tables from the open dataset that are necessary for answering the competency questions. Specifically, we use the following tables from [20]:

1. Film Locations in San Francisco
2. Speed Limits of San Francisco Streets
3. Pavement Condition Index of San Francisco Streets
4. Pedestrian Volume on San Francisco Streets
5. San Francisco City Parks
6. Realtor Neighborhoods
7. San Francisco Housing Inventories of 2005 to 2018
8. San Francisco Muni Stops
9. Park Evaluation Scores starting Fiscal Year 2015
10. San Francisco Tall Building Inventory

The listed tables all come as comma-separated value (csv) files. Therefore, for this scenario, the data component interfaces with files of all the same format. If, for example, some files have tab-separated values (tsv), the data component would need to parse the files according to a tab-separation rather than comma-separation.

3.1.4. The Data Contexts

Recall in Section 2.3.2, we defined a data contextualization to be the process of situating a domain-specific concept with specific data. It establishes how the domain-specific concepts within the domain-specification component are to be grounded in the domain.

In Figure 3, we have the domain-specific concept for a *Maintainable Street*. It contains attributes such as *location*, *Pavement Condition Index (PCI)*, and *speedLimit*. These attributes are mapped to specific attributes of the datasets uploaded. For instance, *location* is mapped to the *Street Name* and *PCI* to the *PCI Score* attributes found in the pavement condition index dataset. Similarly, *speedLimit* is mapped to the attribute *Speed Limit* in the speed limits dataset.

The domain-specific concepts that are defined axiomatically further specify existing domain-specific concepts, so they do not need a mapping of their own. For example, *Dire Street* uses the mapping that is already established for *Maintainable Street* because it is a restriction of that specific domain-specific concept. Therefore, any changes to the mapping of *Maintainable Street* will cascade to *Dire Street*, and any other domain-specific concepts that restrict it.

3.2. Realizing the Knowledge Level of AOMR

In this exercise, the reasoner used is the Pellet OWL Reasoner [21]. Since an OWL reasoner is being used, the conceptualization made by the Theory Level must be translated to an OWL ontology by the Knowledge Coordinator. The process described below is one such translation method, but is by no means the only.

The responsibility of translating to OWL is a significant task, and requires the necessary logic to transform the domain-specific concepts to OWL concepts, object property relations, datatype relations, and data assertion axioms. This is additional to managing the system. Depending on the complexity or detail of the translation to be realized, it might be beneficial to create a separate component which the knowledge coordinator uses. As the Theory Level is independent of an implementation language, we must translate the concepts within Figures 2, 3, and 4 into, in this case, OWL.

The Knowledge Coordinator only translates and uses the domain-specification component(s) that are necessary for the context model selected. For instance, if the context model chosen involves the real estate domain, then the domain-specific concept *Film Site* is not included in the translation process because it is not in the real estate domain, and by extension, not a part of the reasoning process.

Figure 2 shows the domain-specification component for the real estate domain in a general, domain-independent tabular format. For submission to an OWL reasoner such as Pellet, the concepts must be written as OWL assertions. It is an automated process which translates domain-specific concepts to OWL concepts. Attributes are translated to properties of the concept, with the respective typing that is detailed in the domain-specific component.

The domain-specific concepts that are interpretations of an archetype can be viewed as classes in OWL, for example,

```
Declaration (Class (Street))
Declaration (Class (TallBuilding))
```

and each of the attributes within the domain-specific concept can be viewed as data property assertions with the range corresponding to the type declared. For example,

```
Declaration (DatatypeProperty (StreetHasName))
DataPropertyDomain (StreetHasName, Street)
DataPropertyRange (StreetHasName, string)
```

In the situation of a domain-specific concept using, but not restricting, another domain-specific concept, we use an object property rather than a data property assertion. This is due to the domain-specific concept that is being referred to by the attribute already existing as an OWL class. For instance, the domain-specific concept *House* uses the domain-specific concept *Street* as the typing for address. Since *Street* is already an OWL class, we use an object property to relate *House* to *Street* with the relation *HouseHasStreet*. It is then written as

```
Declaration(ObjectProperty (HouseHasStreet))
ObjectPropertyDomain (HouseHasStreet, House)
ObjectPropertyRange (HouseHasStreet, Street)
```

For domain-specific concepts which restrict another domain-specific concept, such as *Busy House*, we define it as an intersection of concepts. For example, *Busy House* is the intersection of *HouseWithMuni* and *HouseWithTallBuilding*. These concepts do not exist explicitly in Figure 2, but correspond to a predicate: *HouseWithMuni* corresponds to all Muni stops which share either an at- or on-street with a house, and *HouseWithTallBuilding* corresponds to the predicate which defines all tall buildings that share an address (street) with a house. They are analogous to an OWL anonymous class. Therefore, we have the following,

```
Declaration (Class (HouseWithMuni))
EquivalentClass (HouseWithMuni, ObjectSomeValuesFrom
    (HouseHasMuni, House))

Declaration (Class (HouseWithTallBuilding))
EquivalentClass (HouseWithTallBuilding,
    ObjectSomeValuesFrom (HouseHasTallBuilding,
    House))

Declaration (Class (BusyHouse))
EquivalentClass (BusyHouse, ObjectIntersectionOf
    (HouseWithMuni, HouseWithTallBuilding))
```

The data that is a part of the context must also be translated to OWL by using data property assertions. As the context includes mappings indicating which attribute is mapped to which dataset, the data property assertions can be written to link the OWL class to the individual.

```
ClassAssertion (Street, 18th_St.)
DataPropertyAssertion (StreetHasName, 18th_St., 18th
    St.)
DataPropertyAssertion (StreetHasPath, 18th_St., 7929)
DataPropertyAssertion (StreetHasPath, 18th_St., 8181)
```

In the above set of data property assertions, we see the name given to the individual, “18th St.”, as well as a list of paths. The

dataset assigns a stretch of road with a specific number. The above assertions assign two of the paths—7929 and 8181—to the street named "18th St.". Multiple paths asserted to the same street indicate that a street has a set of paths.

As the translation uses the dataset to create individuals, it is expected that the data is standardized. This was not the case in the San Francisco dataset: streets were often not recorded with the same value across tables. For instance, in one table, "18th Street" is the recorded value whereas in another, "18th St." is. If it is not identical, OWL will create a new individual. To avoid erronously having multiple individuals, the data was manually standardized. After each of the domain-specific concepts has been translated, the ontology is submitted to Pellet for reasoning.

4. A Proof-of-Concept Implementation of an OMR System

In this section, we provide a proof-of-concept implementation of an OMR system built upon the proposed AOMR. We continue to use the illustrative design scenario and example system for the city of San Francisco described in Section 3.

The proof-of-concept system was developed following the architecture design given in Section 3 and was implemented using Java. It is meant to demonstrate one possible implementation of a system returned from the software engineer to the customer for use by the user.

Figure 5 provides a UML sequence diagram [22] illustrating the user interaction with the proof-of-concept system. The sequence diagram shows how the classes which implement the system components (as given by AOMR) interact to answer a user-submitted query. We point out that in the diagram, some components, e.g., the Archetypes Component, are absent. This is because this sequence diagram corresponds to the interaction a user would have in submitting a query for reasoning, i.e., after the archetypes have already been interpreted to domain-specific concepts, so there is no role for the Archetypes component. The diagram is intended to be read left-to-right, top-down. Progressing down the diagram represents the passage of time. Along the top row is the user and each class that partakes in this sequence diagram. The class is labelled by name, and below it is its lifeline (the vertical dashed line). On top of the lifeline is an activation box, the hollow vertical rectangle. These denote the span of time in which the class is doing something, such as a computation or message call. The solid arrows denote a message call from one class to another, with a cyclic arrow denoting a self-call. The dashed arrows denote the response to a message call.

The interaction begins with a user selecting both a domain and data context on the Reasoning Interface. The Reasoning Interface for the proof-of-concept system is shown in Figure 6. It enables the user to select the context model for which to consider for their query (see ① in Figure 5). The domain and data context are both submitted and transformed into a *i_contextModel* package by the interface. The *i_contextModel* contains only the names of what components are needed for the context model, and not the components themselves. The Data Coordinator then assembles the components listed in the *i_contextModel*–the mappings and the necessary domain-specification component–into the package called *contextModel*. The mappings contain the connections to the datasets declared by the data context. Currently, there is only a single data context: the San Francisco datasets. As the domain-specific concept objects a part of the needed domain-specification component are not saved within the data coordinator, they are retrieved from the interpretations coordinator. As discussed in the design scenario in Section 3, the user is able to select one of three domains: real estate, infrastructure, and tourism. For the purpose of illustration, we suppose the user selects the real estate domain.

Once the Knowledge Coordinator has the respective context model, it retrieves the relevant domain-specific component (see ② in Figure 5). In our real estate domain example, the Knowledge Coordinator communicates with the Interpretations Coordinator to get the domain-specific concepts associated with the real estate domain. For implementation, a repository was made which stored each of the domain-specific concepts: the DS Repository. When the Interpretations Coordinator is retrieving a specific domain-specific concept, it retrieves it from this repository. In this case, the Interpretations Coordinator provides a list of the domain-specific concepts indicated in Figure 2, i.e., those corresponding to the real estate domain. After the Knowledge Coordinator has received the relevant domain-specific concepts, it populates the Reasoning Interface with the queries that are possible for the selected context model as determined by the competency questions for the system (see Section 3). Figure 6a shows this result for the context model that includes the real estate domain context. The list of possible queries is the same as the list of competency questions for the real estate agent viewpoint. Next, the user selects their query and the reasoner to be used when the query is submitted (see ③ in Figure 5).

Suppose that the user selects the Pellet reasoner and submits the query "Tell me about the houses on Castro St.". The Reasoning Interface packages this information and sends it to the Knowledge Coordinator which translates it into a format suitable for the reasoner to obtain and display the results back to the user (see ④ in Figure 5). More specifically, because the user selected Pellet, an OWL reasoner, the Knowledge Coordinator translates the domain-specific concepts and mapped data to an OWL file. This OWL file is submitted to the reasoner, and the respective parts for answering the query are returned to the user. Figure 6b shows the results of this process to the answer to the selected query "Tell me about the houses on Castro St.". As can be seen in Figure 6b, the query result is the data that pertains to the attributes of the domain-specific concept of *House*, with the additional restriction that the address must be on Castro St.. Here we see the importance of the mappings to ensure it meets the needs of the customer. For instance, the real estate inventory dataset contains a *Number*, *Street*, and *Street Type* column. These columns, when put together, make what is commonly understood as an address. However, the attribute of address of the domain-specific concept can only be mapped to one column. As it is mapped to *Street*, we see the value 'Castro' repeated, instead of a full address such as '1102 Castro St'. The

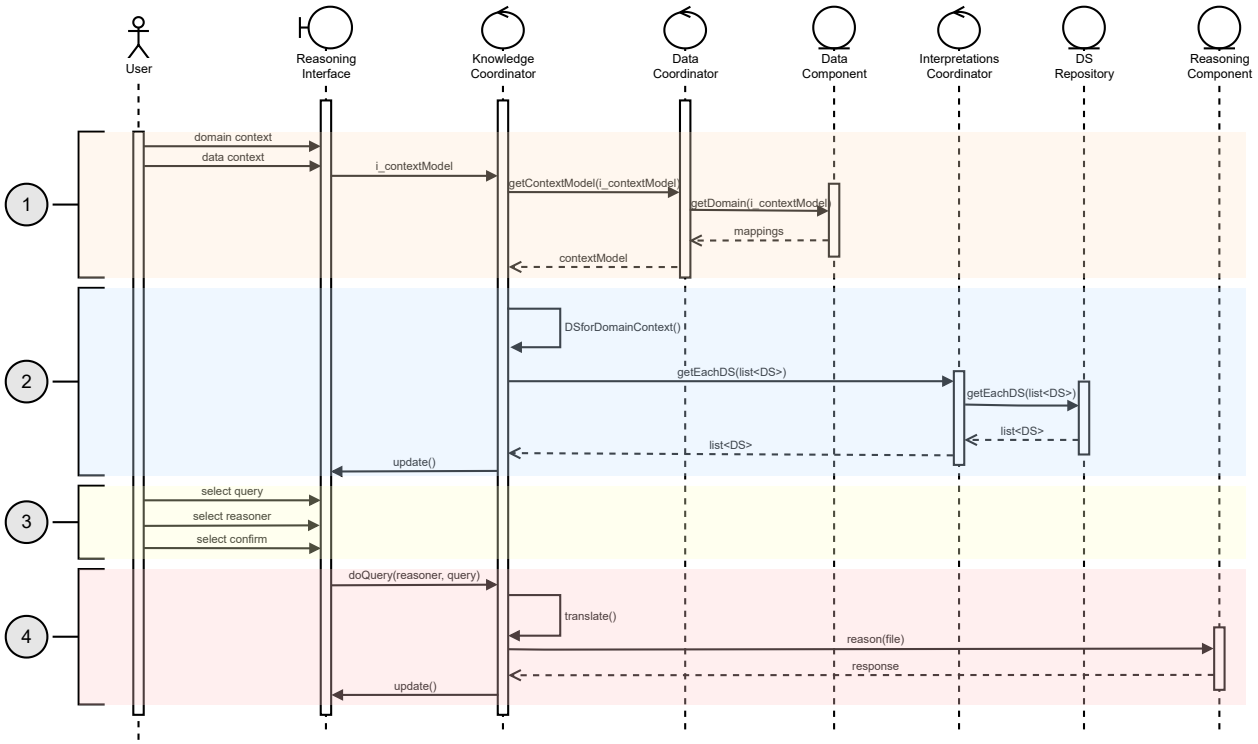


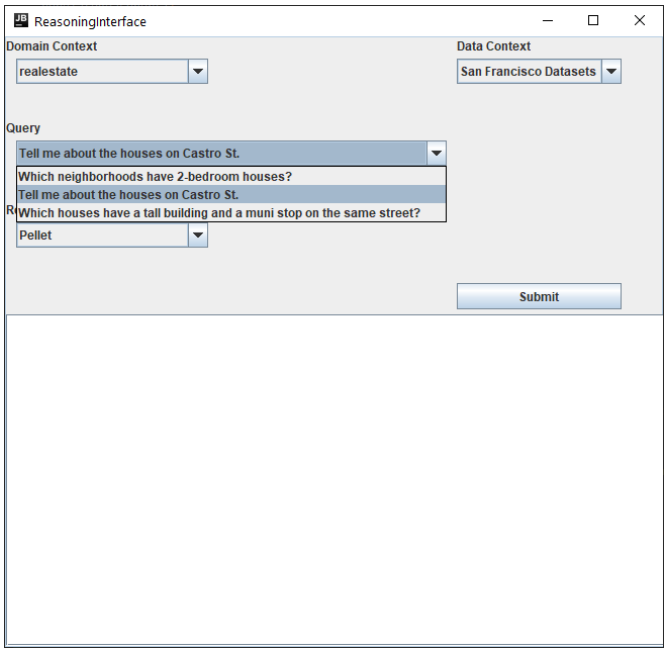
Figure 5: Sequence diagram for the user querying the system.

decision of how to proceed with the mapping is one that must be discussed between the software engineer and customer.

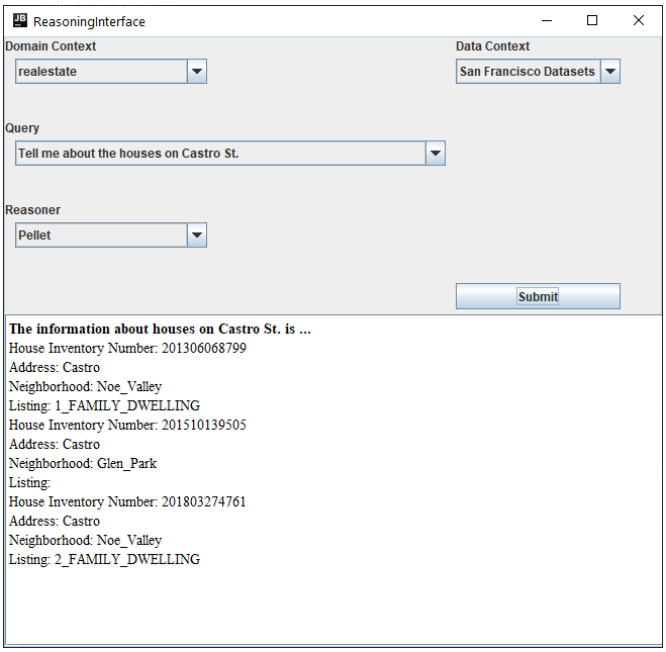
The steps in ①, ②, and ③ can be improved with the addition of more technologies in a more productized version. For instance, using context interpretation technology from natural language processing can enhance the experience by automatically determining either the domain context or data context from the input query rather than having the user select them from a drop-down menu [23]. The query selection of step ② can be enhanced with the implementation of a query processing tool, such as SPARQL [24]. The incorporation of a query processing tool will allow for more general queries to be submitted for answering. For example, instead of only answering the query "Which neighborhoods have 2-bedroom houses?", it could also answer "Which neighborhoods have n-bedroom houses?" for $n \in \mathbb{N}$. Finally, the reasoning tasks of ③ are currently only supported by the Pellet reasoner, but new reasoners can be added. Reasoners such as HermiT, FaCT++, or RacerPro can be implemented for more comprehensive reasoning. However, because we are focused on demonstrating the implementation of an OMR system built upon AOMR, we do not consider these possibilities further.

The proof-of-concept system follows the PAC architecture, with the Knowledge Coordinator controller facilitating the communication with the lower-level controller components. However, there are other design decisions the software engineer must make when implementing such a system. This includes decisions about the internal representation of domain-specifications, archetypes, and context models. It also includes decisions about the format of the data component, and the business logic needed to use the data. For the proof-of-concept system, we chose to use hash maps for these internal representations. Hashmaps were sufficient for the prototype as the dataset was reduced as an effect of standardizing and removing inconsistencies. Although hashmaps were sufficient for the prototype tool, more robust technologies can easily be interchanged. Proper tables can be used in lieu of the hashmaps if the datasets are too large to merit the use of hashmaps and instead require the speed of SQL. Additionally, we created repositories for the archetypes and domain-specification components. The repositories provided the functionality needed to search for specific domain-specification components, domain-specific concepts, or archetypes. As for the data component, because we did not have direct access to a database, the downloaded files were uploaded to a local Derby database. This was done to simulate the interaction of the proof-of-concept system with an existing dataset. We then implemented the data component with the business logic needed to interact with a SQL dataset, such as translating methods to SQL queries. The source code for the proof-of-concept system is available at [25].

It is important to note that AOMR is abstract by its nature (it is an architecture). This means that adopting AOMR requires care and effort on the part of the software engineer throughout the system design activity. When adopting nearly any architecture style, tradeoffs need to be made to realize the functional and non-functional requirements of the system. As demonstrated by the example design scenario and proof-of-concept implementation of an OMR system, the effort required on the part of the software engineer in adopting the proposed architecture and implementing the knowledge coordination is a suitable



(a) Available queries based on the selected context.



(b) Results from the selected query.

Figure 6: Interface for the proof-of-concept implementation of the *San Francisco Ontology* OMR system.

tradoff to reap the many benefits brought forth by the separation of concerns, modularity, and extensibility of the resulting system by adopting AOMR. These benefits, for instance, enable the software engineer to focus on specific contexts rather than large, monolithic ontologies to easily support multi-context reasoning.

5. Discussion

AOMR was motivated by three goals: (1) data transparency, (2) system interactivity, and (3) graceful aging of the system. The data component hides implementation details of the data from the rest of the system. When the data coordinator requires data for the mapping process, it is unaware of whether the data is a SQL database, a csv file, or an abstract data type. It is also unaware of where the data is. For instance, the domain-specific concept *Maintainable Street* utilizes two separate tables in the mappings: the pavement condition index dataset, and the speed limits dataset. However, the data coordinator is unaware of this fact. Thus, the data transparency is achieved by ensuring all implementation details regarding the data is hidden in the data component.

The second goal of system interactivity is not achieved by a single component of the architecture, but rather, by the entire architecture itself. It is a model of the PAC architecture style, which separates the user interface components from the data components, with a controller component that mediates communication. This allows for an interface in which the user can select a context, and for the system to respond by gathering the necessary domain-specific concepts and data. It also allows for the separation of system logic between the agents, such that the knowledge agent, which is responsible for assembling the ontology to be submitted to a reasoner, is unaware of the secrets of the interpretations agent, which is responsible for creating domain-specific concepts from archetypes.

Finally, the third goal of graceful aging is achieved from the separation of concerns between the agents. As it was mentioned, the evolution of the domain may result in new data being introduced, existing data being modified, or changes to the understanding of a concept. If new data is to be added to a dataset which is used in a data context, then no part of the domain-specification components is affected. For example, in the situation that a new data set of *Housing Inventories of 2019 to 2020* were to be released, then the already made data contexts can be updated accordingly. The data that populates the domain-specific concept of a *House* can be updated to align with the 2019 and 2020 inventory. However, if the customer wishes the new data to be used in its own data context, then a new data context must be established. The currently existing data context which uses the 2005 to 2018 housing inventory will remain unchanged, but a new data context, which uses the 2019 and 2020 inventory can be created. The data coordinator will be aware of both existing data contexts, and allow for a user to interact with the system using whichever data context they desire. The separation ensures that changes made to the data component will not cascade to, for instance, the domain-specification component.

In the case where a new domain is to be added because of new customer requirements, a new domain-specification component must be created, and new archetypes can be added if needed. The modification of an existing domain-specification component does not cascade into another, as they are designed to be independent from one another. Extending the knowledge coordinator to translate to a new implementation language is

also a simple task because of the platform-independent design approach [26]. Software engineers are free to choose the ontology modelling or implementation language, and extend the knowledge coordinator with the necessary translation methods. Regardless of what the change is, the location to which the change must be made is straight-forward. Data changes effect the data component, concept changes effect the archetypes and domain-specification components, and translation changes effect the knowledge agent. Thus, the maintenance of the system is straight-forward from an evolution perspective.

Although the illustrative scenario proposed in this paper utilized disjoint domains, AOMR does not impose such a restriction. It might be the case that a combination of domains is necessary to answer a question, that is to say, answering the question requires more than one domain-specification component. For instance, to answer "*Which houses are on a street that does not need maintenance?*" we would need to use the knowledge from the real estate domain and the city infrastructure domain. In particular, the information of houses from the real estate domain, and the information of maintainable streets from the city infrastructure domain. However, to do so would require that the combined domain remain consistent. For instance, both the real estate domain and city infrastructure domain have a domain-specific concept for *Street*. To combine these domain such that the domain-specific concept *Street* does not become bloated with attributes, and remains consistent, is not trivial. This particular problem is referred to as *ontology alignment* and is an extremely active field of research for ontologies [27].

Industry 4.0 involves ecosystems each of which is comprised of a large number of devices and uses a high volume of data. This data is to be contextualized which allows the needed behaviour of each of the devices or involved agents to be inferred. These devices/agents are recognizable and smart as their decision is context-related [28]. It is this context-aware capability that gives the main characteristic of smart products and services emblematic of the Industry 4.0. Designing systems that are context-aware and that can employ the appropriate context for each decision or action is one of the major challenges of Industry 4.0 [13]. While the example that we used in this paper is about interpreting city data within several contexts, it can easily be related to the challenges that we find in Industry 4.0 situation. For instance, the manufacturing in Industry 4.0 relies on data, collected from various sensors and from the requirements of each work-pieces, to make appropriate manufacturing decisions or actions. This same data for one manufacturing device is, for example, to be interpreted in a context of milling or grinding and for another agent is to be interpreted in the context of preventive maintenance. The data is the same, but the domain of interpretation of that data is different from one device or agent to another. This generation of actionable information from data within a domain is analogous to what we have illustrated in our example; the city data set is interpreted in several contexts (e.g., geographical, ecological, or city-infrastructure maintenance) to generate information specific to each of these perspectives.

6. Related Work

The architecture proposed in this paper, AOMR, provides the conceptual base for an OMR system. In this section, we evaluate the existing literature regarding both the architecture of an ontology-based system and multi-context ontology-based systems, as well as how these systems are used in Industry 4.0. We also investigate the methodology in which an ontology-based system is created to better contextualize where in the design process AOMR, or any comparable architectures, can be used.

The work in [29, 30] is formative research in developing an architecture for an ontology-based system. In these works, an MVC architecture is proposed that separates the ontology from the reasoning system. However, there is limited functionality in providing a multi-context experience and in establishing a context model. This is due to the restriction of only a single view component in an MVC architecture. AOMR extends this research, by incorporating the PAC architecture, to address both of these limitations.

The definition of a context by Dey et al. [31], which is cited heavily in the research of context-aware systems, is the information that situates an entity. In context-aware ontology-based systems, the context manifests itself using information such as time, location, or user. For instance, in [32], Gu et al. present a system motivated by a smart home scenario. Examples of contexts are: in which room a specific user is and at which time of day they are in the room. Depending on the context, the system will behave differently. In [33], the context is also the data environment of the concept or entity. More specifically, it is the relationships between the object and the children, ancestors, data properties, and object properties. In [34], Haruna et al. present a survey which evaluates several domains and collects the contexts that are relevant for these domains. Examples include time, nationality, and weather being a part of the context for a place. Often, context-aware systems are used in pervasive computing, such as in [35, 36], where data is constantly being collected through the environment, typically into a smartphone or other portable device. In [37], Othmane et al. present a recommendation system that uses multiple contexts to provide its recommendation. Although there are multiple contexts, such as the desire context, the goal context, and the belief context, they have little to do with the definition of context used in this paper. For Othmane et al., the contexts provide rules and ways of generating consistent recommendations from the data. In this paper, a context is a way of understanding both the data itself and the concepts of the domain. Finally, in [38], Akhtar et al. present a framework for multi-context systems for the purpose of defeasible reasoning. However, the use of context in this work includes various domains, and much of the work attempts to rectify any inconsistencies. AOMR is intended to be applied to a single domain, such as a city, where inconsistency is not a primary concern. In this work, an alternative understanding of context is used. Instead of the information that situates an entity being data such as location or time, it is instead the set of mappings of specific datasets to domain-specific concepts. A change of context model in AOMR can result in a domain-

specific concept such as 'Street' being interpreted as an Int (e.g., "7929") instead of as a String (e.g., "18th Street"). A change in context model implies a change in how the domain is to be understood, rather than only an update of the data that situates an entity.

AOMR presents a unique architecture for the ontology-supported system to facilitate different understandings of a context model. The system presented by Mtiiba et al. in [39] addresses the problem of data transparency and 'multiple facets' of data that AOMR also addresses. However, AOMR is an architecture whereas the system presented by Mtibaa et al. is an implemented system. Systems such as [32, 40] that adhere to the definition of a context model as solely the data that situates a concept are ordinarily designed as a multi-layered system. In such a system, there exists a general ontology on the upper layer, and several domain-specific ontologies in the lower layer. In the upper layer, the concepts provide a shared vocabulary each of the lower layers will use, such as location and time. The lower layers are specific domains, such as a house or car. Most notably, these ontologies are written in OWL. The implementation of a system following AOMR is not only independent of ontology implementation, meaning it is not necessarily reliant on an OWL ontology, but are able to provide multiple understandings of the generalized concepts of the domain. The archetypes are most similar to the upper layer ontology concepts in that the archetypes provide the concepts which conceptualize the domain, and can be used in various domain specification components. Unlike the existing context-aware systems such as [32], the archetypes of AOMR are instantiated into domain-specific concepts which allow for a single archetype to be understood in several ways. This approach is similar to that of Benslimane et al. in [41]. Where they present a language for writing a multi-context ontology in Description Logic (DL). With it, they can state that a student is a person who has a student number (context 1) or is a person who is enrolled in courses (context 2). This is similar to instantiating different domain specification components for archetypes.

The incorporation of ontologies, or ontology-based systems, is becoming a more common topic of research in the Industry 4.0 setting. In [42], Jasko et al. discuss ontologies in the field of manufacturing execution systems (MESs). Several requirements of the ontology-based system are outlined, including the incorporation of data, modularity, and context-awareness. They define the system as being context-aware if it is able to factor its environment into its decision making processes. Jasko et al. present a table that lists how existing ontologies that support Industry 4.0 perform with respect to the mentioned requirements. Most, if not all, do not achieve more than 2 of the 6 requirements found in [42]. Therefore, more effort needs to be put on building systems that meet more of these requirements. AOMR is an architecture for ontologies that promotes data transparency, modularity of the system, interoperability among domains, and adaptability to new data sources or domain specifications. These are four of the requirements listed by Jasko et al. in [42]. An example of a designed and implemented ontology for the Industry 4.0 setting is shown in by Ramirez et al. in [43]. There they design and build an ontology for describing an extruder machine. When designing it, they followed the NeOn methodology. Although able to gather and form a proper list of competency questions to guide the design, they were unable to consolidate five different dimensions (contexts) that the questions could be classified into. Thus, they were challenged to create an ontology that was able to be reused to answer the questions for any of these dimensions. AOMR allows for this type of system by representing each dimension as one of the domain specification components. The parts that were reused by Ramirez et al. are reminiscient of the archetypes in the AOMR system. From these two cases, it is apparent that the problems associated with Industry 4.0 ontologies, specifically reuse, data transparency, and modularity, are all properties exhibited by AOMR.

It is also important to note that AOMR is intended to be used within a system design methodology. AOMR is an architecture for an ontology-supported multi-context reasoning system, which is most comparable to context-aware systems (such as those described above), and ontology-based systems. Examples of methodologies for designing an ontology-based system are [44] and [45], which have been evaluated in [46]. The methodologies all begin with the problem definition, determining the feasibility of the project, and ending with an implementable ontology-based system. Although ways in which to critique or evaluate an ontology-based system are often discussed, the process by which to *develop* an ontology-based system is not. We propose that, much like the process of following an architecture when designing software in software engineering, an architecture should be followed when designing an ontology-based system. AOMR is one such solution where the specific purpose of the system is to support reasoning over multiple context models.

The systems for automatically retrieving classes of information from natural language text by processing them are referred to as Information Extraction (IE) systems. These systems have several application domains. In [47], we find that they cover natural language understanding, speech recognition, syntactic analysis, semantic analysis, pragmatic analysis, and speech synthesis followed by major natural language processing applications which include machine translation, information extraction, information retrieval, sentiment analysis, question and answering chatbots. Recently, Ontology-based Information Extraction (OBIE) systems have become a major trend in IE processing [48] and they have a common architecture that can be identified from their implementation. The common architecture of OBIE systems is given in [48] and it corresponds to a specific case of AOMR with only one context.

7. Conclusion and Future Work

OMR systems have potential applications in numerous emerging technological and industrial paradigms, that require the ability to incorporate and utilize multiple context models. The context models allow for various domains of a world to be conceptualized. Current architectures for ontology-based reasoning systems allow for single contexts to be conceptualized,

and so using multiple contexts would require multiple implementations. AOMR allows for multiple contexts to be conceptualized in a single system, allowing for simple interactions between the user and the system. Depending on the context model that is selected, different types of queries can be answered.

We foresee that the issues that we have isolated and prescribed to a domain suitable for OMR systems to rise in need as smart systems and Industry 4.0 becomes more widely used. These issues included numerous participating agents that each have their own unique view, multiple heterogeneous data sources, and a conceptualization of the domain that is used for reasoning. We present AOMR as an architecture so that as these new systems are created, they have an architecture to guide them.

In this work, we propose AOMR, an architecture for an OMR system where data transparency, interactivity, and graceful aging is necessary. PAC is an ideal architecture for an interactive system, and the modular nature of PAC promotes a maintainable system where each acting agent hides certain implementation details from one another. This results in a system where the evolution of data is isolated to a single component of the system, and does not cascade to several others. It also results in a system where maintaining or extending the system is a straight-forward task. Finally, the use of archetypes and domain-specification components allows for a customer to easily determine if the proposed domain fits their requirements.

The architecture was demonstrated using a proof-of-concept realization of an envisioned OMR system using open data sets for the city of San Francisco. Three domains were evaluated, and the example scenario demonstrated how the various context models can be chosen to influence the reasoning capabilities of the system. There is future work in the further productization of the prototype tool by incorporating technologies such as SPARQL or NLP, to enrich the context model selection and query submission process. With the use of these technologies, it is hypothesized that the context model can be inferred by the submitted query, and that richer queries can be submitted. Additionally, research in combining several domains is of great interest; this would require extensive use of the existing research in ontology alignment [27]. Although this paper introduces AOMR and explores how it can be used to design an OMR system for a city, we intend to apply this to a proper Industry 4.0 or Healthcare 4.0 setting. This task requires a clean and accessible set of data, like what was used for the city, as well as proper competency questions. The goal is to demonstrate applicability to these domains. Finally, we foresee the development of a proper pipeline to semi-automatically ingest data and, with entity-matching technology, create domain specifications or archetypes. The user can then either accept or reject the proposed domain specification components.

Acknowledgements

The authors thank the anonymous reviewers for their thorough reviews that greatly improved the quality of the paper. This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the grant RGPIN-2020-06859.

References

[1] V. R. S. Kumar, A. Khamis, S. Fiorini, J. L. Carbonera, A. O. Alarcos, M. Habib, P. Goncalves, H. Li, J. I. Olszewska, Ontologies for industry 4.0, The Knowledge Engineering Review 34 (2019).

[2] A. S. Ullah, What is knowledge in industry 4.0?, Engineering Reports 2 (8) (2020) e12217.

[3] P. Bouquet, F. Giunchiglia, F. van Harmelen, L. Serafini, H. Stuckenschmidt, Contextualizing ontologies, J. Web Semant. 1 (4) (2004) 325–343. doi:10.1016/j.websem.2004.07.001.
URL https://doi.org/10.1016/j.websem.2004.07.001

[4] N. Guarino, D. Oberle, S. Staab, What is an ontology?, in: Handbook on ontologies, Springer, 2009, pp. 1–17.

[5] F. Giustozzi, J. Saunier, C. Zanni-Merk, Context modeling for industry 4.0: An ontology-based proposal, Procedia Computer Science 126 (2018) 675–684.

[6] C. Reinisch, M. J. Kofler, W. Kastner, Thinkhome: A smart home as digital ecosystem, in: 4th IEEE International Conference on Digital Ecosystems and Technologies, IEEE, 2010, pp. 256–261.

[7] V. V. Estrela, A. C. B. Monteiro, R. P. França, Y. Iano, A. Khelassi, N. Razmjooy, Health 4.0: applications, management, technologies and review, Medical Technologies Journal 2 (4) (2018) 262–276.

[8] T. Gu, X. H. Wang, H. K. Pung, D. Q. Zhang, An ontology-based context model in intelligent environments (2020). arXiv:2003.05055.

[9] A. Firat, S. Madnick, F. Manola, Multi-dimensional Ontology Views via Contexts in the ECOIN Semantic Interoperability Framework, Working papers 4543-05, Massachusetts Institute of Technology (MIT), Sloan School of Management (May 2005).
URL https://ideas.repec.org/p/mit/sloanp/17070.html

[10] J. C. Augusto, Ambient intelligence: Basic concepts and applications, in: J. Filipe, B. Shishkov, M. Helfert (Eds.), Software and Data Technologies, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 16–26.

[11] N. Henze, P. Dolog, W. Nejdl, Reasoning and ontologies for personalized e-learning in the semantic web, Journal of Educational Technology & Society 7 (4) (2004) 82–97.
URL http://www.jstor.org/stable/jeductechsoci.7.4.82

[12] T. H. D. Varun Grover, General perspectives on knowledge management: Fostering a research agenda, Journal of Management Information Systems 18 (1) (2001) 5–21. arXiv:https://doi.org/10.1080/07421222.2001.11045672, doi:10.1080/07421222.2001.11045672.
URL https://doi.org/10.1080/07421222.2001.11045672

[13] B. Chatterjee, N. Cao, A. Raychowdhury, S. Sen, Context-aware intelligence in resource-constrained iot nodes: Opportunities and challenges, IEEE Design & Test PP (2019) 1–1. doi:10.1109/MDAT.2019.2899334.

[14] E. Bertino, A. Kundu, Z. Sura, Data transparency with blockchain and ai ethics, Journal of Data and Information Quality (JDIQ) 11 (4) (2019) 1–8.

[15] L. Wang, Heterogeneous data and big data analytics, Automatic Control and Information Sciences 3 (1) (2017) 8–15.

[16] J. Coutaz, Pac-ing the architecture of your user interface, in: Design, Specification and Verification of Interactive Systems' 97, Springer, 1997, pp. 13–27.

[17] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, Z. Wang, Hermit: an OWL 2 reasoner, Journal of Automated Reasoning 53 (3) (2014) 245–269.

[18] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, Y. Katz, Pellet: A practical owl-dl reasoner, Journal of Web Semantics 5 (2) (2007) 51–53.

[19] M. Stocker, M. Smith, Owlgres: A scalable OWL reasoner., in: OWLED, Vol. 432, 2008, pp. 1–10.

[20] DataSF, San francisco open data, https://datasf.org/opendata/ (2020).

[21] L. Clark & Parsia, Pellet, https://www.w3.org/2001/sw/wiki/Pellet (2020).

[22] K. Qian, X. Fu, L. Tao, C.-w. Xu, Software architecture and design illuminated, Jones & Bartlett Learning, 2010.

[23] G. Di Fabbrizio, S. S. Bharathi, Y. Shi, L. Mathias, Context interpretation in natural language processing using previous dialog acts, uS Patent App. 14/283,017 (Nov. 26 2015).

[24] J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of sparql, ACM Transactions on Database Systems (TODS) 34 (3) (2009) 1–45.
[25] A. LeClair, J. Jaskolka, W. MacCaull, R. Khedri, AOMR prototype, https://github.com/aLeClair/aomr-prototype (2020).
[26] B. Selic, On software platforms, their modeling with UML 2, and platform-independent design, in: Proc. of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2005, pp. 15–21. doi:10.1109/ISORC.2005.40.
[27] M. Ehrig, Ontology alignment: bridging the semantic gap, Vol. 4, Springer Science & Business Media, 2006.
[28] I. Bisio, C. Garibotto, A. Grattarola, F. Lavagetto, A. Sciarrone, Exploiting context-aware capabilities over the internet of things for industry 4.0 applications, IEEE Network 32 (3) (2018) 101–107. doi:10.1109/MNET.2018.1700355.
[29] J. Jaskolka, W. MacCaull, R. Khedri, Towards an ontology design architecture, in: Proc. of the 2015 International Conference on Computational Science and Computational Intelligence, CSCI 2015, 2015, pp. 132–135.
[30] J. Jaskolka, W. MacCaull, R. Khedri, Towards an architectural framework for systematically designing ontologies, Tech. Rep. CAS-15-09-RK, McMaster University, Hamilton, ON, Canada (Nov. 2015).
[31] A. K. Dey, G. D. Abowd, D. Salber, A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications, Human–Computer Interaction 16 (2-4) (2001) 97–166.
[32] T. Gu, X. H. Wang, H. K. Pung, D. Q. Zhang, An ontology-based context model in intelligent environments, arXiv preprint arXiv:2003.05055 (2020).
[33] X. Xue, C. Jiang, Matching sensor ontologies with multi-context similarity measure and parallel compact differential evolution algorithm, IEEE Sensors Journal 21 (21) (2021) 24570–24578.
[34] K. Haruna, M. Akmar Ismail, S. Suhendroyono, D. Damiasih, A. C. Pierewan, H. Chiroma, T. Herawan, Context-aware recommender system: A review of recent developmental process and future research direction, Applied Sciences 7 (12) (2017) 1211.
[35] D. Ejigu, M. Scuturici, L. Brunie, An ontology-based approach to context modeling and reasoning in pervasive computing, in: Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PerComW'07), IEEE, 2007, pp. 14–19.
[36] G. Specht, T. Weithoner, Context-aware processing of ontologies in mobile environments, in: 7th International Conference on Mobile Data Management (MDM'06), IEEE, 2006, pp. 86–86.
[37] A. B. Othmane, A. G. Tettamanzi, S. Villata, M. Buffa, N. Le Thanh, A multi-context framework for modeling an agent-based recommender system, in: 8th International Conference on Agents and Artificial Intelligence (ICAART2016), 2016.
[38] S. M. Akhtar, H. M. Ul Haque, Contextual defeasible reasoning framework for heterogeneous systems, in: Context-Aware Systems and Applications, and Nature of Computation and Communication, Springer, 2020, pp. 16–30.
[39] A. Mtibaa, F. Gargouri, A multi-representation ontology for the specification of multi-context requirements, in: International Conference on Signal-Image Technology and Internet-Based Systems, Springer, 2006, pp. 259–269.
[40] J. Kim, K.-Y. Chung, Ontology-based healthcare context information model to implement ubiquitous environment, Multimedia Tools and Applications 71 (2) (2014) 873–888.
[41] D. Benslimane, A. Arara, G. Falquet, Z. Maamar, P. Thiran, F. Gargouri, Contextual ontologies, in: International Conference on Advances in Information Systems, Springer, 2006, pp. 168–176.
[42] S. Jaskó, A. Skrop, T. Holczinger, T. Chován, J. Abonyi, Development of manufacturing execution systems in accordance with industry 4.0 requirements: A review of standard-and ontology-based methodologies and tools, Computers in industry 123 (2020) 103300.
[43] V. J. Ramírez-Durán, I. Berges, A. Illarramendi, Extruont: An ontology for describing a type of manufacturing machine for industry 4.0 systems, Semantic Web 11 (6) (2020) 887–909.
[44] Y. Sure, S. Staab, R. Studer, Methodology for development and employment of ontology based knowledge management applications, ACM Sigmod Record 31 (4) (2002) 18–23.
[45] S. Sarnikar, A. Deokar, Knowledge management systems for knowledge-intensive processes: design approach and an illustrative example, in: 2010 43rd Hawaii International Conference on System Sciences, IEEE, 2010, pp. 1–10.
[46] R. Dehghani, R. Ramsin, Methodologies for developing knowledge management systems: an evaluation framework, Journal of Knowledge Management (2015).
[47] R. S. T. Lee, Natural Language Processing, Springer, Singapore, 2020, Ch. 6, pp. 157–192. doi:10.1007/978-981-15-7695-9_6.
[48] D. C. Wimalasuriya, D. Dou, Ontology-based information extraction: An introduction and a survey of current approaches, Journal of Information Science 36 (3) (2010) 306–323.