ABSTRACT

Title of dissertation:	DYNAMIC ONTOLOGIES THAT ENCODE AND MANAGE RELEVANCE IN CONTEXT AWARE SYSTEMS
	Nicholas Gramsky Doctor of Philosophy, 2018
Dissertation directed by:	Professor Ashok Agrawala, Department of Computer Science

Context aware systems, to date, tend to fall into one of two categories: domain specific or generic across multiple domains. Domain specific systems are single-use instances – that is, establishing the ability to manage context for an additional domain necessitates the creation of an additional system. Authors of such systems should instead strive for generic ones. Generic context management systems require a generic modeling and context delivery system.

Previous research has shown that generic context aware systems prove to be quite dynamic through their use of ontologies. These ontologies, however, are very rigid in nature, requiring additional software to mature and manage instantiated models, filter relevant information, or pre-cache information. The result is users who wish to use generic systems must encode relevance across ontological models, filters, and newly created external software with each re-use in order to manage context manipulation at run time.

Through the design and implementation of Rover3, while leveraging the concept of an Automatic and Dynamic Information Model (ADIM) methodology, we outline what we believe how context aware systems should function. By providing a framework to encode relevance within ontologies, we minimize the way to present and consume relevant information. Our context management framework uses dynamic ontologies to deliver relevant information to users striving to achieve goals for any given situation.

Walking through an accident response case study we showcase the aforementioned features of Rover3, showing how such incidents can benefit from context aware systems. The value of Rover3 is expressed through an extensibility study where efforts to expand existing ontological models are compared between Rover2 and Rover3.

This dissertation presents:

- The notion of relevant context and how it can be managed at runtime through a generic context aware system.
- The required primitives and rules for modeling any generic situation.
- The Automatic and Dynamic Information Model (ADIM) methodology, how one can encode relevance in a general information model, and exhaustive grammar and rules for this version of ADIM.
- The Rover3 system and its application of ADIM, showcasing how it provides a generic framework to model and manage context that does not require any additional software.

DYNAMIC ONTOLOGIES THAT ENCODE AND MANAGE RELEVANCE IN CONTEXT AWARE SYSTEMS

by

Nicholas A. Gramsky

Dissertation 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
Doctor of Philosophy
2018

Advisory Committee:
Professor Ashok Agrawala, Chair
Professor Min Wu, Dean’s Representative
Professor Adam Porter
Professor Alan Sussman
Professor Atif Memon

Dedication

For my children - Miles, Bryce, and Rainer. For my wife Denise. And anyone else who finds this work moving.

Acknowledgements

First, I want to thank my advisor, professor, and friend in Dr. Ashok Agrawala. I had a very rough first semester at UMD. After ignoring the advice of my assigned advisor I decided to take a seminar taught by Dr. Agrawala. His motivation, inspiration, and free thinking allowed me to blossom. This was the start of a long and exciting journey where computer science, philosophy, and all aspects of science were discussed. During this time, you challenged me in ways few ever have. You planted a few seeds and helped them flourish to the ideas in this dissertation. As a result, I now view the world and problem solving in a whole new light thanks to your time and energy. I only hope I give back to a few the ways you have given to so many students such as myself. The collaborative environment you foster within the MIND lab is great. My only regret was I did not spend more time with the lab and fellow students. I hope we continue to have coffee and lengthy discussions like those had during this exploration.

My lab/class partners over the years have done so much more than help me complete projects, they taught me about software dev, how to think, and other views on life. Robert Gove, Ken Knudsen, and Preeti Bhargava are a few I worked at length with. My time with each of you was beneficial both in terms of the innovative challenges we solved as well as the personal and technical growth I underwent with each of you.

To my managers/supervisors that allowed me to run to College Park for day classes I thank you. You didn’t have to allow me the flexibility to do this but your willingness to allow me to grow was pivotal in my ability to continue my career during this journey. Bill Milligan, Mauricio Martinez, Dan Domingue, Rizza Eklund, and Joe Walker, I thank you profusely for allowing this to take place.

To my friends who never knew how painful it was to be asked “are you almost done yet?”, thank you for pushing me. The list is lengthy but there were a few of you who just never let up are Michelle Castro, Ines Mason, Blair Lockamy, Ted Moore, Andre Williams, and many others.

To my dad and my grandfather, thank you for instilling in me the value of hard work. Seeing my father work tirelessly every day with ever-increasing high standards eventually caused this to be pre-wired in my brain. My deceased grandfather, Ed Crescenze, who still holds a UMD baseball school record, thank you for teaching me to never give up, aim high, and work hard.

Though their interactions with my schoolwork was mostly in the form of sitting on my lap and asking if they could push keyboard buttons, messing up my office, or distracting me, my kids were one of my biggest motivators to finish. So many times I wanted to give up. Quit. Return to a life without school. Each time that thought cross my mind I cringed at the story I would have to tell my kids “I was almost there, but it got too tough and I decided to quit…”. That has not been, nor will it ever be how I operate. In my most difficult periods, I thought of them and the lesson I would them by finishing.

Lastly, but surely not least, absolutely none of this would be possible without my loving and supporting wife, Denise. It was her encouragement, dedication, sacrifice, and fortitude that allowed me the time and energy to embark on and complete this long journey. I thank the world I have been graced with her presence. Her tolerance and support of this effort is just one of many examples of how loving and committed of a life partner she is. I hope these words provide some semblance of thanks for the many days she handled the kids, let me sleep in after an all-nighter, took care of affairs around the house, and dealt with my grouchiness, all while I toiled away at this degree. Readers of this dissertation should be prouder of her role in the production of this document than they should be of my writing of it.

Contents

1 Introduction

1.1 Motivation

Any situation itself has a lot of information. The simple act of me typing this paper has a wealth of information that surrounds the scenario. I am in my house, in the den, at 6:12 AM, typing with low light, I have 10 windows open in the background of my computer, and the kids are still asleep. I am sitting in a metal chair, wearing a sweatshirt, and the inside temperature is 70 degrees. The walls in the den are painted golden amber, the doors are glossy white and the newly installed windows have yet to be painted. As everyone else is asleep in my house and outside activity is mostly limited to nature, the amount of ambient noise is very low. In fact, the sound of me typing on the keyboard is the loudest thing within 50 meters. This list of surrounding environmental data goes on and on and I am only discussing the act of typing this paper. Most of this information, however, provides me with no value in my ability to write the paper.

If we consider a more interesting situation, say a two-car accident, there exists an even larger set of information; information we can actually capture and use to help us if we are motivated to help rectify the situation. We have both cars, the driver of each car, any passengers, the driver's health records, the driver's police records, the history of the car, the location and time of the accident, and so on. As the event unfolds there will be emergency personnel supporting those involved, the possible need to locate a hospital to transport anyone that is hurt, as well as additional police force if the accident was crime related. Amongst other dimensions, information can be classified in two areas – static information and changing information [1]. The color of the cars, the history of each person, the location of the nearest hospitals, and driving ability of each driver is static and will not change within the temporal context of the situation. Other information however is changing almost rapidly. Blood pressure of those involved, safety of the cars involved, position of nearest ambulance, and capacity of a hospital and its ability to take on new patients given their ailments could be changing by the minute or even second. Despite the volatility of the information, those involved in trying to assist with the situation will want to constantly react and re-evaluate their actions at any moment in order to ensure any safety issues are mitigated.

Data for any scenario can be found in many places. Internal systems and databases may hold health information for individuals. Different databases hold police and criminal records. As the Internet of Things (IoT) continues to increase the number of smart devices around us to 75 billion devices in 2020 [2], the rate upon which information is created and collected grows daily. Cameras are aplenty, providing both closed circuit systems and public view of almost any point on the globe. Police cars in the US are outfitted with cameras continuously reading license plates, allowing officers to passively record information as they drive. The billions of cell phones double in function as both a communication device and a multi-sensor device. With the proliferation of smart devices to include smart watches and other wearable, the multiple sources to passively collect data has continued to exponentially grow. Knowing 'where' to simply gather information itself can prove to be quite the challenge, much less efficiently fetching data for processing purposes

Meta-data about the information itself is quite valuable as one evaluates any single piece of information. Despite her best intentions, my mother's account of my health records as a child are likely overshadowed by the records the various doctors I visited. An eyewitness to a crash might provide an account of where one car was when it hit another and how fast it was going, but a camera with a full account of the incident would provide images that allow for an exact re-creation of the event that could include exact speeds, times, and placements to the inch. Well calibrated GPS sensors could provide an even more accurate description of speed and location than an objective camera, resulting in yet an even more reliable account of the incident.

Despite the wealth of information found in any situation, not all data is really relevant for any particular goal. Revisiting the situation where I am working in my office, if I want to ensure I am able to work productively the fact that the tone of white paint on the doors is less important than the fact that the kids are asleep. The amount of ambient noise surrounding my office is likely relevant to my ability to concentrate, but only on certain tasks. I have found that some tasks, such as software coding or mathematical problem solving, are actually bolstered by music of certain types. That said, outside of using the computer that contains references material for this task, consuming this information does not assist in my ability to accomplish the goal of writing this dissertation.

Likewise, emergency responders reacting to the car accident scenario above will want to resolve the situation quickly. For example, mitigating any risk to those not involved in the accident, treat any injured, and clean up the scene are all possible goals of such emergency responders. Immediate and optimal response of such a situation is one where complete and relevant information can help expedite the necessary actions to take. For example, when dealing with an injured person in the car accident, their professional history might not do much, but knowing that a bleeding passenger has blood clotting issues might help prioritize who to treat. Various involved parties will have goals relative to this accident. Some will want to help the injured, others are looking to clean the scene, and others will look to ensure no foul play was had. Each member or group of members will need different pieces of information pertaining to this incident. Collecting and sorting through all information to determine the best course of action however can be a challenge for humans. The wealth of historical information as well as current and changing information can best be handled by context aware systems that employ Human-Centered Computing [3] (HCC) principals.

Delivering relevant information on a situational basis is accomplishable through context aware systems. Such systems should deliver any information that can assist in changing the state of the situation to a more ideal state; whether the information be complete or simply a reference to information that needs to be obtained. The ability to do so, however, typically involves domain specific models or software that need to be created each time a new circumstance warrants such. In doing so, relevance is typically spread throughout the model, filters, and delivery software. Such solutions have opportunities to lower the friction on having relevant information ready for any given situation.

In this dissertation, we present what we believe a context aware system to be through the definition and creation of a dynamic information ontology and the delivery of the Rover3 system. A dynamic information ontologies is the mechanism that encodes relevance within an information model. As relevance is situationally based, every possible thing that can exist in an ontology, the way any two (or more) things relate to one another, and all attributes for both can be defined as a function of the situation. We are presented with the ability to define an expansive information model that defines how any part of it can assist with any encoded situation. The result is the eradication of information filters and the ability for an appropriate framework to instantiate only relevant information for consumers as situations arise.

Rover3 is a context aware system designed to process, store, handle, deliver and explore the information surrounding given situations, providing human beings the assistance they need to quickly model, sort through, and receive relevant information. We believe this classification of information is critical to assisting humans resolve situations. Rover3 consumes dynamic information ontologies and automatically delivers relevant information by adopting a novel ontological modeling methodology we call the Automatic and Dynamic Information Model. This allows Rover3 to store and pre-cache any enriching information, and deliver information that assists in resolving any situation the users of the system are presented with. Rover3 is a general context management system, meaning it processes, delivers, and manages contexts of all types. As the system consumes dynamic information ontologies, Rover3 is the last context management system one would need to develop. Adding a new domain simply means creating an additional or expanding an existing ontology. Adding or modifying relevance is a simple model adjustment. This removes the need to write any additional software. The result is users who need to consume relevant information just need to create ontologies for their domains and let Rover3 handle the model instantiation and information management.

1.2 Current Context Aware Systems

The notion of context aware systems [4] [5] [6] has been around for many years. A quick survey today shows a majority of the research [7] [8] and applications for context aware systems are limited to mobile applications [9] [10] and users [11]. Such systems require software and coding that is very specific to the domain itself. Evolution of the domain requires changes to the modeling and the software managing the context. Attempts to generalize the approach can be seen through Rover [12] [13], Rover2 [14] and a handful of ontological modelling techniques that Rover2 has used [15] [16]. Ontologies can help minimize this effort as the modeling aspect of the context can be abstracted away from the software management aspect of a context aware system. Yet all of these approaches need something else to truly manage relevant information as the collection of information evolves.

Rover is a context-aware, middle-ware platform that provided relevant information for any individual or entity. Through a set of APIs, Rover provides a framework to provide the context of an entity on demand. Rather than create many separate service calls and iterating over multiple APIs and data endpoints, Rover provides a middle-ware solution that standardized how such relevant information queries and data sources were handled. This set of libraries was the first of its kind to template and abstract such calls. Services and queries were generalized through the framework, empowering developers. By providing filters or methods to supply relevant data on demand, applications and end users quickly get information that would aid in the specific goal or situation an entity was presented with.

Rover2 provides a more extensible framework through the use of ontologies. For the first time, multiple information taxonomies across multiple domains can feed Rover2's generalized information modeling framework. Utilizing the Rover Context Model Ontology (RoCoMo) [15] [16] as a generic ontology for any information set, the information framework defines the rules of how a context aware middle-ware system should model any data. A set of primitives were defined as the base set of objects any dynamic information system could use when modeling and working with data. Entities and relationships found in existing ontologies are instantiated within Rover2, assigning them to events and enacting activities on the instantiations as needed.

While both Rover and Rover2 define the rules for context aware computing and generalize the methods and data structures for dealing with information from any domain, there is still a bit of encoding that is required to model and work with information [17] [18] [19]. Subject Matter Experts (SMEs) are required to define their respective information taxonomies and capture how different entities within their field are defined and how they relate to one another. Yet this is not enough. Before one can use the data from Rover2 one is required to write extensive code or software whenever a new domain is to be used. Though the types of calls and modeling are extensible enough to adapt, the refinement of data and definition of relevant information still requires external functions outside of the framework. Filters exist to provide only the relevant data, but that requires one to apply the specific filter. Single pathways exist that define how information is instantiated and used at run-time for any one situation. If system designers want to manipulate context models as information or situations evolve, they are forced to write software that executes the logic for such manipulations. Ideally the framework for a context aware information management system is extensible enough to remove the need for coding as it should only require a SME to craft his ontology itself and not write custom code for context management.

We argue that context aware systems can be developed for any domain without the need for software to define relevance or manage. Rather than delivering context for a particular use case, limited applications, or needing additional software to manage context, we believe that with the correct modeling framework in place any end system, user, or application can receive relevant information with no additional computation or management. This unbounded use extends beyond mobile applications. Simply calling it middle ware is rather limiting. As it can provide information discovery capabilities, direct navigation through the UI console and RESTful API's, our proposed solution allows for information discovery and management.

Lastly, there is nothing in any of the aforementioned systems about managing relevant information as the situations they represent change. Situations evolve over time and the relevant information for any one situation changes. Whether we model the domain with an ontology, or as a part of the software that comprises the context aware system itself, something must evolve the relevant information model as the situation being modeled evolves. All of the above system require additional software any time we can construct a new way a situation can evolve.

1.3 Human Knowledge is not Static

Despite relevant information being provided to end users and systems, many systems handle real-time decisions about information in code [20] [7] [14] [21] [8] [22] [23]. We argue there are actions, explorations and filters that should be applied at run-time to information systems that become aware of such information. To date, those actions, discoveries, and filters are facilitated through code and not solely a function of a model itself. We also argue that the modeling of data should not be limited to the structure of the entities and relationships themselves, but a bridging of the gap between thought and reasoning needed to take place as situational modeling [24] is to occur. Relevant information is a combination of known, preconceived models, whatever data exists for any scenario, and the appropriate filters that should be applied. The result is information systems that spread the notion of relevance across the models, filters, and software. Software must orchestrate actions across all such systems in order to consistently manipulate what is considered relevant information.

Many general models have been considered when discussing context aware systems [4] [11] [25]. Key-value models are simple in nature and look to define uniform structures for knowledge definition. UML and object-oriented modeling provide a little more structure to the context that can be delivered with the latter allowing for interfaces to the data itself. Markup models allow for tagging of data and are seen in the efforts of the semantic web and RDF.

In this dissertation, we show that an ontological model is the best mechanism to deliver relevance as they allow for a wide array of things to be modeled, including concepts and relationships between models. Yet all current implementations of ontological models are limited in that the relationships, interfaces, or structures are locked in; any variability is only handled as makes decisions about how to instantiate the taxonomy either in code or through an external exercise. As is, the reasoning around when requires the various components of today's information systems are handled outside of standard context brokering systems, much less the structure of the information model itself.

We believe that ontologies [26] themselves need to be extended beyond their conventional approach to date. The rules around how to form an ontology need to be as dynamic as human thought and modeling. Relationships in the real world that tie entities together are dynamic and vary based on given situations. Understanding how humans model their thought process is key. Situational awareness/handling can also be modeled despite the fact that actions humans take are dynamic and vary based on what information is presented to them at a given time. Though typically stored in logical computer programs, we believe this general relevance can be captured in an ontological model, managed and delivered through a proper context management system, and no longer distributed across multiple aspects of an information system.

1.4 Contribution

In this dissertation, we deliver a framework that delivers relevant information for any given domain with minimal effort. Through the delivery of a new ontological modelling system that captures relevance and a context-aware system to facilitate this methodology, we believe this is easily achieved.

In this dissertation, we:

- Define the notion of managing relevant context at run-time.
- Define what it means to model any given situation. Through an extensive exploration in the various components needed for context modeling, we look to provide all of the necessary components to do so in a way that best represents the reality around us.

- Define the notion of a dynamic information taxonomy. Existing ontologies and modeling methodologies are quite static in nature. We define what it means to define a dynamic information taxonomy, how to encode relevance in an information model, as well as show how this method of modeling is a step closer in mirroring the physiological habits about how humans learn and represent their reality. This will be accomplished with the Automatic and Dynamic Information Model methodology (ADIM) described in chapter 5. We then look to implement this and show a case study, highlighting the flexibility on onboarding the system as well as the unique ways to model and process real-time relevant information.
- Design, implement, and deploy the framework described above through the delivery of Rover3. Rover3 is an advancement of the existing Rover and Rover2 systems. Like its predecessors, Rover3 is a context-aware system that provides relevant information as needed for any given scenario. Unlike the previous systems however, Rover3 requires no coding for use across unlimited domains as relevance is encoded in the ontologies it consumes through the use of ADIM. Through the use of visualization tool and web interfaces, users can experience the expansion of models, context delivery, and how information discovery can take place as well as augment the model instantiation process.
- Demonstrate the Rover3 system by walking through an example public safety situation, providing context and relevant information to consumers as needed. This case study will also show how the Rover3 system allows

for information discovery ease of use and added context delivery for any given situation.

- Perform an extensibility study between Rover3 and Rover2, showing the relative use and scalable uses of such a context aware system.

1.5 Organization of this Dissertation

This dissertation is organized in multiple chapters. In Chapter 2 we outline the various mechanisms that are required to model information and situations. Chapter 3 explains how we believe a context aware system should be built, defining design tenets to solidify this approach. The notion of a dynamic information ontology is introduced and explained in chapter 4. The Automatic Dynamic Information Model is a dynamic information ontology that is introduced in chapter 5. The Rover3 framework and system are described in detail in chapter 6. We explore the complete ADIM grammar in detailed in chapter 7. We then model and walkthrough the evolution of an emergency response case study in chapter 8 that makes full use of the ADIM and Rover3 technologies. An extensibility study is explored in chapter 9, where the effort to add additional domains is compared between Rover3 and existing systems. We conclude in chapter 10 with future work.

2 Context, Relevance, and Ontologies

2.1 What is Context?

According to Merriam-Webster, one of the definitions of the word context is:

> "the interrelated conditions in which something exists or occurs
> : environment, setting"

This definition of context is quite broad as it suggests any and all information that exists and relates to any one thing or setting would suffice as context. Users that would want to consume 'context' by this definition would need to filter out that information they might prefer, ignoring all other inter-related information.

The broadness of the above definition places quite the burden on anyone or anything that would use context. As the interrelated conditions for any set of entities is vast as time evolves and the environment potentially and perpetually changes, one would have to manage the context and only use / consume what is desired. It is then more appropriate to suggest that context is a subset of all interrelated things that relate to one another for a particular environment or setting. Yet if we want to extract value from any of this information we would have to define why we only need a subset and what drives the subset.

Maybe more appropriate to the world of computer science, Dey [27] defines context as:

> "..any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."

Other attempts [28] [29] [30] have outlined the various definitions of context and how the definition of context is often relevant to the domain being discussed. Dev's definition better refines the notion of 'environment' in the previous definition and we should consider the concept of the situation as that which bounds context. Context could be considered any relevant information that an end user or system would find useful and applicable for a given goal or a given situation, where relevant information is information that is useful to attain any number of goals for any given set of situations.

One important aspect of Dev's definition are the first two words: "any information". Just because a consumer or computer system does not have or know about any specific piece of information does not mean such unknown information is not considered to be a part of the context of the situation. Situations are not static either. Like mathematical functions, they can easily evolve over an infinite number of discrete values. For example, a person who is hurt in a hospital might transition from critical, to serious, to stable conditions over the course of their stay and the interactions between the entities in that situation may vary. Thus, the definition of context might be extended to include all possible interactions and entities that have yet to be considered or realized in a situation, as well as how those interactions vary as a function of any one given set of information.

2.2 Ontologies as the Method for Modeling

As defined and discussed above, the definition of context, in its simplest form, is a collection of information. While many methods to manage context [1] [4] have been proposed, ontologies [31] [32] are one such method. Looking at definitions and properties of an ontology, we outline why it is the mechanism of choice for modeling context.

According to Wikipedia [33] the definition of an ontology in the context of computer science is:

> "a model for describing the world that consists of a set of types, properties, and relationship types."

Working in the field of Artificial Intelligence, [34] Gruber used ontologies for general knowledge sharing when designing artificial intelligence and defined them as:

> "An ontology is a description (like a formal specification of a program) of the concepts and relationships that can formally exist for an agent or a community of agents. This definition is consistent with the usage of ontology as set of concept definitions, but more general."

Commenting further on Gruber's definition we can define some of our own terms and clarify this definition. Rather than concept we will use the term entity. In an attempt to identify any component of an ontology we will use the term 'primitive' to mean either an entity or a relationship as each of these will share some common traits across both. Further simplifying the definition, one can say an ontology is a collection of primitives put together as needed by a pre-defined set of possible rules. These rules might be better known as a language.

Ontology languages adhere to a standard grammar. It is this grammar that facilitates the rules and jargon about how they are constructed. This jargon, among other things, differentiates relationships and entities, provide ways to assign attributes, and expand entities to more specific types. It is the toolset to enable an information model. The rules of the grammar ultimately dictate what can and cannot exist in an ontology and it defines how to build a specific ontology. The grammar can vary and is a function of the ontological framework that is used. Not all ontological grammars are the same and thus the grammar and ways one can construct discrete ontologies is a function of the ontological framework that is used. It is the grammar and the rules about the grammar that authors need to consider as they look to create an information model. The right framework with the right ruleset can allow one to model anything.

It is this property of extreme flexibility that make ontologies the ideal mechanism to model context. The ability to model any collection of entities and relationships allows one, with the right framework and grammar, to present an information model that is generic and broad enough to model any situation. Such ontological frameworks are extensible and abstract enough to allow for any domain, collection of entities, and/or situation. With ontologies, we can instantiate some or all parts of an ontology into a software system. Doing so allows consumers of ontologies to instantiate a version of reality that the ontology supports, ensuring the structure of the information is consistent across applications.

While typically there are more than two, the two primitives that exist in every ontological framework are relationships and entities. Relationships are simply exercised rules that explain how one or more entities relate to or interact with one another. These rules stitch together the many ways the components of the ontology interact and help provide the rules for any possible instantiation of the model. A relationship makes a connection between one or more entities, though they commonly connect two or more entities. A relationship can simply define an interrelation from one entity to itself. The relationship can be cyclical and define a state change for another entity. You can, for example, feed yourself.

Both of these terms, entity and relationship, can be decomposed into collections of other entities and relationships. The human body, for example, is a collection of limbs, torso, head, mind, organs, and many other components. The actual body is one entity and each sub-component is another entity. Limbs, for example, have other sub-components like joints, flesh, and skin. Each of these are composed of smaller components such as cells, molecules, atoms, etc. The relationship 'is composed of' is one such relationship that builds all of these entities up to the model of the body itself. Decomposing entities through a collection of refined entities and relationships is a further refinement of the model itself, suggesting that an ontology can be a collection of models at any given point of decomposition.

Both primitives can further be described through the use of attributes. While relationships provide the rules for how entities engage with one another, evolve, or interact, attributes are the characteristics of individual primitives that serve to refine a particular instance of any primitive. An example of this could be the color of the entity 'car', the weight of the entity 'frame', the amount of torque on the relationship 'turns', or the time of the relationship 'started'. Attributes can either be linked to entities through relationships "entity X has attribute Y with value Z" or the ontological framework can support the assignment of an attribute. An example of the latter would be a JSON object with any number of key:value pairs that represent attributes.

Models that are created in ontologies can be generic or refined as needed. Consider an accident response example, a generic accident might involve the relationships and entities required to identify who was involved in the accident, any possible additional people who could end up getting hurt, evacuation plans, danger radius, and response teams. More specific accidents may involve car accidents or chemical accidents. These models might bring with them more specific sets of relationships and entities or refine that set, removing some generalized ones. For example, a model for a two-car accident by default might not include the need to identify evacuation routes.

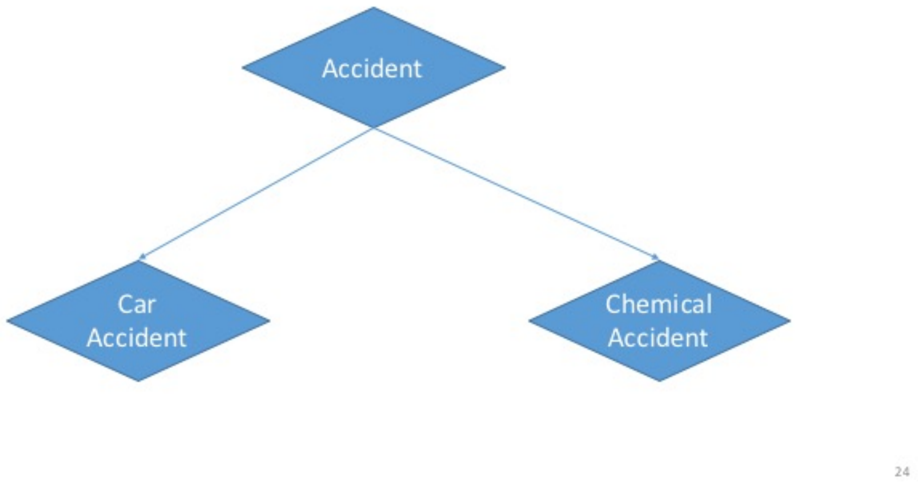


Figure 2.2.1 – Sample Hierarchy of Generic and Specific Accidents

This decomposition and refinement of models can continue. The 2-car accident we discussed earlier would be an even more refined model of a generic car accident. A 2-car accident has different connotations and needs regarding cleanup versus, say an 80-car pileup. As a result, the model for both accidents would be different and contain with it different sets of default relationships and entities for each accident type. They both, however, might inherit goals of a generic ‘car accident’ model which might be to ensure the safety of all those involved.

There are multiple ways to use ontologies and the use is independent of the domain the ontology is facilitating. Ontologies can simply be an information model for something (person, computer program, etc.) that one wishes to represent or reference. Such ontologies can act as a reference to an information model for the purpose of having all or parts of the ontology instantiated for discrete uses. An example of this might be an ontology of the human body where a software program instantiates a copy of that ontology to perform some computations on a simulated version of a human body that may exist in the real world. Other ontologies are living copies and are expanded and populated with discrete information. An example is the use of ontological storing of information in the Semantic Web. In this implementation, new sites have the ability to add on to the ever-expanding information store that is the reference to the entire internet. It is thus important to realize that the term ontology can be used to reference an abstract ontology or an instantiated one.

The use of ontologies in this dissertation will be of the first example. We will refer to an ontology that was or is being produced to model and standardize the information structure of a domain. Once created, the intended use will be such that some or all of the ontology can be instantiated in order to perform some function on the instantiated copy as deemed relevant. As we author an ontology, we are building an information model and using the ontological language we propose to do just that.

2.3 Relevance

We have mentioned before that relevant information is information that is useful in the attaining of a goal. Our claim is that any system or entity, be it biological or mechanical, is looking to either stabilize or modify some environmental baseline around or about itself. In the first case, something has happened that disrupted what is considered to be acceptable – some state has changed, some entity is not as it should be and the desired action is to return the environment to the same state again or a new acceptable state. In the latter, a new baseline is desired and some method of mutating the environment is desired. In each case, there was a general end goal of achieving a desired state.

Relevance is simply the notion of how a select set of primitives (amongst all possible primitives for a given domain) are connected for achieving the goals of various situations. As we build models, our aim should be to ensure the relevance for any and all situations across the primitives is complete. That is, if we turn to the ontology to dictate all possible entities and relationships that can ever exist, we will be presented just that. As we go to instantiate the ontology for a discrete implementation, we can use a relevance filter and get only the subset of the model; a relevant subset that will assist us in achieving a particular goal.

Let's return to the example of a two-car accident and specifically look at how one might utilize an ontological model for a person in such an example. There may be many reasons to model a person in such a situation – the people responding to the accident will create pre-conceived mental models or computer systems might need to model the environment. First let's assume the default ontological model for a person that includes all possible relationships is that which is show in figure 2.3.1. In this example, we see that a person might own a car, be the driver of a car, has personal attributes, health records, professional records, and a social network. However, a person does not always have all of these traits. Not every person owns a car or home and the professional credentials may vary or be missing altogether in some circumstances.

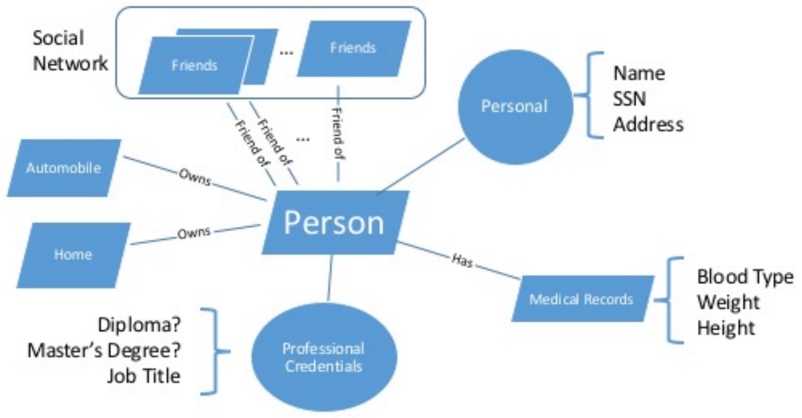


Figure 2.3.1 – Sample Complete Model for a Person

Additionally, this general model contains things that need not be fully used in the situation of a car accident. The fact that they own a house, their profession credentials, and social network have no bearing on the actions one responding to the accident will take. Those responding to the accident have a goal in mind to resolve the accident. The previously mentioned information does nothing to assist them in their ability to resolve the accident. We can conclude the general model of the person that incudes these bits of information is a bit excessive. A more relevant and filtered model of a person for a car accident might be that in figure 2.3.1.2 where the relevant portions of the model are blue and irrelevant information is greyed out. This might be the exact model we would instantiate if we were to instantiate a subset of this generic model. Within the larger framework of the 2-car accident model, each person involved might be modeled as such. The expectation that the person might be hurt warrants including medical records for the individual. We need to know their personal information as we do for any car accident, and knowing what cars they own helps with any insurance issues.

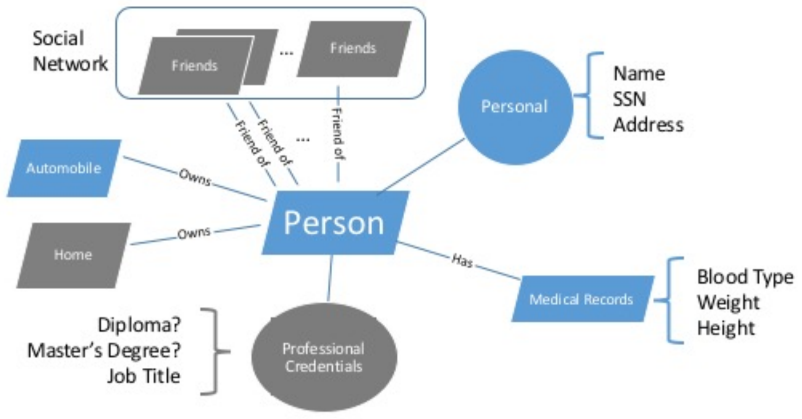


Figure 2.3.2 – Default Model for a Person in a Car Accident

As the car accident unfolds and that instantiated model of the situation expands, it might be the case that one of the cars involved in the accident is holding explosives. In this evolving situation, it might be beneficial to know the social network of the driver and the passenger of that car to see if there are any terrorist ties within their friends and colleagues. This would require a slightly larger instantiated model for that individual as noted in picture 2.3.1.3

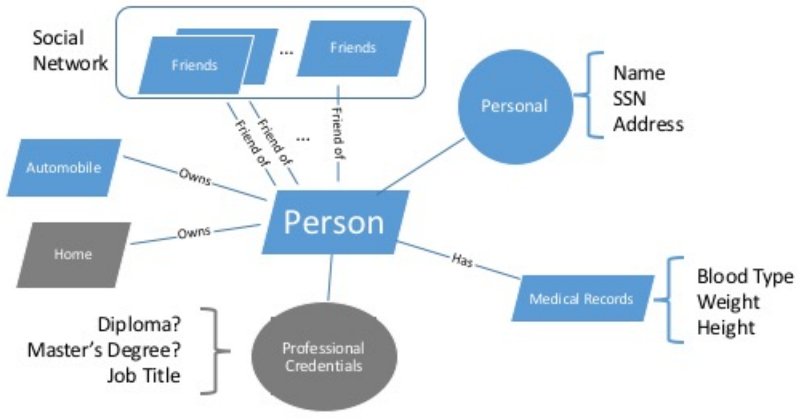


Figure 2.3.3 – Model of a Person expanded to include Social Network

Initially these models are all of a generic person. There was a base set of components we needed (personal information) to help identify the person. The template for the model included all possible models one could instantiate. The situation upon which we needed to instantiate each model is what drove the actual composition of the model in the run state.

The ontological models we look to build, however, need to have the ability for one to encode relevance within them. When we say we encode relevance within a model we mean we capture what components of a vast model would ever assist in achieving particular goals. Any one domain might contain an enormous collection of entities and relationships, yet every single component at comprises the ontology is likely not relevant for every situation. Encoding relevance into a model is the placement of the rules that dictate what is instantiated and mutated at run-time.

As we choose to instantiate parts of an ontology (be it entities, situations, or entire domains) we should include all entities, attributes, and relationships that could ever exist for that desired situation we are modeling in our instantiated ontology. In doing so, we should also capture the relevance for all possible goals. We will revisit this two-car example again and see how we might encode the relevance within the ontological store itself and we discuss how to place relevance logic within every aspect of an ontological model.

2.4 Primitives Required to Model Context

The ability to provide proper context for any given scenario relies on proper modeling and mechanisms to deliver the relevant parts of that model. Defining a context modeling ontology that is truly abstract is key. Our framework is one that needs to be truly flexible, so our primitives need to be as basic as the raw components of knowledge itself. We believe that means defining a modeling methodology that incorporates all aspects of this chapter.

The RoCoMo context model [35] defined what is necessary to model relevant information for a given situation. In order to effectively model information, RoCoMo defines the necessary primitives for an information management system. The primitives outlined in this system include Entity, Activity, Relationships, and Events. Any piece of information that needs to be handled or modeled is contained in a primitive. Multiple primitives are related and joined in an effort to model given situations. Though sufficient in finding relevant context of a given situation, the system needs a fair amount of software development to process a given situation. For example, when requesting relevant information, applications needed to define their own relevance filters. These filters then provide the data that is relevant to their situation at hand. Both the models used and the instantiated models were available for the end system, though the decision to use a model was limited in the situation that was modeled. We argue that the ontology itself can do this filtering through the instantiation of the model as it is needed in real time.

Though other models have been proposed [36], we believe the RoCoMo [16] model closely aligns with the way to model and abstract reality. It does, however, require some improvements in order to better represent the world around us with minimal data loss. We propose extending this model, expanding on Entity and adding two new primitives – Situations and Goals. The resulting primitives and definitions we propose any context model should include are:

- **Entity** – Real or virtual. The basic building block for any model. Entities can be permanent or temporary. Examples include people, inanimate objects, data structures (ex: emails). Within the Rover3 ontologies that we will utilize, all physical entities will leverage the notion of location. This location can be both specific (lat/long or street address for example) or relative (within a 2-mile radius of College Park).
- **Activity** – Carries out the physical or virtual manipulation of information or entities. Generates information, takes time, and results in a change of the system, specifically modifying attributes of some primitives or introducing or removing primitives.
- **Goal** – Final state in an activity. Goals also server as the desired end state for a situation. Goals are either a modification of the current context or the stabilization of context that is no longer desired by entities. Situations exist

until the corresponding goal is met or no longer a valid goal. A goal does not have to accompany an activity, but every situation must have a goal.

- **Relationship** – Describes how primitives relate to one another. These relationships have attributes and information as well and can be derivative or transitive. Relationships allow entities to be hierarchical in nature as they can aggregate more refined entities into larger, abstracted versions. For example: The body entity might decompose into varies entities like limbs, mind, torso, feet, hands, etc. All of the entities could be bound together with the 'is composed of' relationship. Each of those could also be broken down into another set of cells that also are 'composed of'. Different situations/instantiations would define where in the hierarchy of the model we need to be, or rather, what entities are considered relevant.
- **Situation** – This is what a person or system is handling. This can be one or multiple events. Situations define the boundaries of the use of the system. A situation can be very brief (cell phone needing current location) or extensive (extinguishing a prolonged building fire). This, like all primitives, situations can be organized as a graph or collection of primitives. Rover2 has situations, but they are not primitives but rather templates of primitives. It was very close in nature though being a primitive we can build hierarchal situations just the same as any other collection of primitives.

As information relevance is situationally influenced, the need for a situation primitive is apparent. It is the situation primitive that really frames the context. As many context-aware systems are written for particular domains or require coding to help define the domain, the situation primitive is the mechanism that helps bridge an end application to this framework. Existing ontologies and primitives can be re-used over and over, and new situations can be added to any collection. Each situation has a distinct goal that helps end applications. As a result, the model is complete and does not need any outside tooling in order to describe how it works or how it is used. The exception is the code needed to instantiate models, help propagate data, and continually evolve what portions of the model are considered relevant as situations evolve. This will become clear as we discuss the necessary framework to deliver this in the next two sections.

2.5 Ontologies that Contain Relevance and Instantiation Logic

So far, we have described the mechanisms and primitives required to build an information model. We have decided that an ontological modeling approach is extensible enough to capture any combination of entities in the world and is best suited to deliver relevant context to aid consumers sorting through information to achieve a goal. With a goal to embed relevance in the ontological model itself coupled with relevance being dictated by a given situation, we must look to establish a way to ensure the ontological model has a mechanism to only instantiate what is relevant for a given situation.

These ontologies, however, can be quite vast. As we expand the composition of an ontology itself (say, as the domain expands or we more accurately model a collection of entities and by result, make the model larger), we are faced with a dilemma of how to determine what portions of the ontology to instantiate at any one given time. Furthermore, we need to determine this subset to instantiate rather quickly and efficiently – some of the situations could require immediate resolution. In either way, we need to minimize the effort to gather only the relevant portions of the ontology. We need to optimize both the accuracy of the relevance and the speed of gathering such relevant information. We thus must turn to mechanisms within the ontology to automatically determine what is relevant. Without this, the performance of a user will suffer as they sort through either a larger than necessary information model or the necessary steps to filter a generic ontology.

Having the situation primitive as a building block is not enough to mechanize the instantiation of relevant portions of any ontological model. Within a domain, any subset of primitives can or cannot be relevant for any one given situation. As our goal is to not have a software program per domain to manage what is continually relevant or an external filter to determine what components of an ontological store are relevant, we need to place the logic that determines what is relevant into the ontological store itself. In order to do so, one must utilize an ontological grammar that contains the instantiation logic dictating which components of the ontological store are considered relevant for any subset of situations.

Revisiting the example model of a person and the two-car accident from section 2.3, there are places where the relevance could be encoded in the ontological store itself. In figure 2.3.2, the relationships and corresponding entities that we considered relevant (owing the automobile, having medical records, and personal information) could only be instantiated if the situation 'accident' were to exist. Attributes of these relationships could state that if a situation of an accident were to exist in an instantiated context model then the relationships and corresponding linked entities themselves are considered relevant and too should be instantiated.

In figure 2.3.3 we found the person to be a dangerous person, so the relationships and entities that comprise the social network extension to the model might be instantiated if the situation is of ‘terrorist situation’ or attribute of ‘considered dangerous’ is tagged to the person themselves. Situations need not be discretely known or instantiated to drive the further instantiation or modification of an instantiated ontological model. A situation can be inferred through the existence of known information – a man who is present in a simple situation (let’s say, ‘having a conversation’) but found to be bleeding might infer an ‘injury’ situation or a man driving recklessly with a bomb in his car might warrant a ‘public safety’ situation. An ontological framework that encodes relevance in an ontological store needs to embrace logic to modify the ontological model based on the current known situation – we must have the ability to evolve what is considered relevant with every information update we receive. That that logic must allow for updates from multiple information sources and not just the existence or omission of a discrete situation.

This instantiation logic must be programmable – situations evolve and our relevance filter within the ontological store needs to accommodate for this. This concept is important for the figure 2.3.3. In that example, a subset of the information store had already been deemed relevant (medical records, automobile, personal information) when the situation was a two-car accident. As more information was known about the real-world situation, it warranted a larger subset of the ontological store to be instantiated. Common software external to the ontological model needs to both instantiate and monitor the ontology itself in order to continue to provide a relevant subset. After all, an ontology is not a software system, it is just an information store. A properly designed context aware system leveraging an ontological language and framework can perform the run-time actions described.

3 Context Aware Systems

As described in the last chapter, an ontology alone cannot instantiate relevant information models for a given situation. Software systems such as context aware systems are one mechanism that can consume both abstract and instantiated ontologies. An appropriately constructed generic context aware system can deliver relevant context to any type of consumer (person, machine, etc.) and does not require business logic to exist in code. We now look at what a context aware system should look like so we can deliver relevant information with the purpose of helping humans and/or systems to achieve any given goal.

3.1 What is a Context Aware System?

Asked what context aware systems are today and the academic world seems to think this concept is mostly confined to mobile computing [37], IoT [38], and ubiquitous computing [30]. Common approaches and modeling have been levied in these attempts in order to produce common contextual data for these domains. Data such as location, activity, and surroundings are often returned and many have worked to provide a common platform to deliver such information. There have been attempts to create context aware systems for other, specific applications [39] [40, 12]. Each of these systems tend to define new models for those domains as well as created certain APIs for the end application at hand. With the exception of the Rover systems and very few others [41] [28], none of these systems handle, model [42] [43] [25] and process context in a generic form that is domain agnostic.

Those systems that strive for a generic context model or ontological framework still require a bit of software to deliver relevant information after information is instantiated [44] [45] [46]. We argue that a real context aware system does not need to be limited to any particular domain nor require additional software to extract, manage and deliver relevant information. As context is any information that is relevant to a query, thought, situation, group of people, physical reaction, or mental state to name a few, a context aware system should be one that can facilitate any domain as well as the ability to deliver what is relevant. Such a system must have the ability to model both the domain and what 'relevant' means. Through the correct abstractions and decomposition of the primitives that make up knowledge, beings, and real-world representation, a context aware system can provide what is relevant for anything.

3.2 Tenets of a Context Aware System

One goal in this dissertation is to design and build the Rover3 system in order to store and deliver relevant information as we believe it to be. Users or systems of Rover3 can use it as a data store in any capacity – either augmenting external data stores or completely relying on Rover3 as their database of choice, both to aid in the accomplishment of any generic goal. Before we start to lay out how Rover3 works, let's define how we believe a context aware system should be constructed. The tenants of a context aware system as we believe them to be are:

- **Extensible modeling framework** – Models are re-usable and can be re-used and extended as needed. Using ontologies is an example of this tenet.

Not doing so will require users to create redundant information models each time they wish to add a new domain to the

- **Provide relevant information** - Relevant information is, of course, relevant to the current situation at hand. The system should not be confined to one domain but rather have the ability to provide any kind of relevant information. Relevance also requires knowing all information available. Rather than just returning what an end application asks for or believes it simply needs, a context aware system should have all relevant information that has yet to be asked. The adage 'we don't know what we don't know' is applicable here; the system should provide data deemed useful based on models and not only what an end has requested or acquired to date. Information discovery is as important as information retrieval, and context aware systems should not be limited to only provided what one asks for. Without this tenet, consumers are left to sort through every bit of context.
- **Minimize abstraction of reality** – Functions and software programs should look to update and fetch data as much as possible. Programs that help model discrete data add noise to the stored data. We look to have the model itself define how information is used, not software external to the model.
- **Proactively fetch data and expand models at run time** – Information should be available when humans or systems need it. If the information needed could have been pre-cached, a later need for a query or computation is removed and the data is delivered and consumed faster. Likewise, as situational exploration continues and possibilities are explored, models

should be expanded as the models dictate. This requires the models themselves to self-prune and expand as well as direct the system to take action in order to pre-fetch data or expand models.

- **Provide for hierarchical context** – Information and the effort to abstract information needs to be hierarchical. As needed, data should be enriched or simplified. The modeling of data needs to do the same; the framework should have the ability to model something as generic or as detailed as possible and the abstraction of that model should be a part of the framework itself.
- **Remove the need for manual context management** – This is the big tenant for management of context. End users should simply receive what is needed or relevant. Though interactions can be provided to help guide the system further, the management of filters, data expansion, and model instantiation should be minimal or completely effortless. Simply put, context should evolve at runtime.
- **Help humans make decisions** – The system should be an aid for humans that are looking to resolve a situation. The relevant information should be valuable enough to help do just that. As the system is here to *help* the human, the outputs cannot be without human intervention. Mechanisms need to exist that allow humans to modify what is considered relevant, by either pruning or adding information that the system might not otherwise.

- **Adopt features of HHC systems** – As noted in [47], HHC systems “can be defined as ‘the development, evaluation, and dissemination of technology that is intended to amplify and extend the human capabilities to:
    - perceive, understand, reason, decide, and collaborate;
    - conduct cognitive work;
    - achieve, maintain, and exercise expertise.’”

Context Aware systems should be developed using these tenets. Not doing so will result in consumers to exert extra effort to manage context and spend energy creating redundant models and software systems.

4 Dynamic Ontologies

Embracing the tenants of a context aware system in chapter 3, specifically the HHC tenant around the notion to "perceive, understand, reason, decide, and collaborate" [47], the providing of relevant information should be as empowering as possible. As Rover2 [14] noted, context aware systems need to be free of any one domain or use case [15]. The ontologies typically used in context aware systems and pervasive computing systems [48] are quite static in nature and do not allow the structure of the model to change. We must look to improve the notion of how one simply constructs ontologies for such uses.

It is thus important to understand the difference between the templated model that can be used, the difference between the instantiated model, and how the evolution of the instantiated model processes. The dynamic ontology is the collection of all possible models for a given domain (or domains) that has the relevance logic used to instantiated relevant subsets encoded within it. One can think of this as a template store – every entity, every possible relationship, activity, situation, etc. is stored in this ontology. As situations unfold and context aware systems instantiate them (a situation is required for any active, instantiated model), templates from the ontology are instantiated into working models. Logic, encoded into the template store along with all of the primitives, is what helps drive the evolution of the instantiated model. Context aware system then deliver the relevant information to end users via these working models.

We now discuss the notion of a dynamic ontology, how such a concept embraces the HHC tenant, and the overall benefits of using one.

4.1 What is a Dynamic Ontology

We define a dynamic ontology as a standard ontology that allows for:

- **Encoding of variable relationships** – Rather than simply stating how entities always relate to one another, a relationship can be variable or may not exist based on a situation or collection of primitives. The variability itself is also captured within the ontology in some manner.
- **Variable hierarchical entities** – Ontological models can be extensive, hierarchal models. In such situations entities can be related to entities, which in turn can be related to more entities. These can be hierarchal or sprawling in nature. Despite this, the filtering or pruning of any one model is done outside of the model in information systems to date. A dynamic information ontology should allow for the encoding of these expansion rules to exist within the model itself.
- **Collection of actions that goal-driven entities may take** - Relevant information suggests there is a use for the data with an end goal in mind. The model should support the ability to drive towards goals. Additionally, the ontology should allow itself to be embedded with rules that allow for automatic model expansion; the framework itself has a goal to continuously provide possible relevant information.
- **Relevance to be defined at run-time without effort** – True 'relevance' for any situation need not be manually defined through filters or additional models or templates. Nor does the ability to delivered relevant information require additional code. Relevance is the intersection of a properly defined

model and the instantiation of the model itself, allowing for only relevant primitives to be used and created. The rules of the collection of model should such that the existence of entities and relationships and information is acquired and models are instantiated, only relevant information is placed in the instantiated model.

4.2 Why do we need a Dynamic Ontology

Reviewing previous context aware systems that leverage ontologies, we see areas to further abstract such a framework and minimize the need for additional software to deliver relevance. Rover2 [14] utilized ontologies in single and cross domain uses. Relevance filters were created for given situations and this helped deliver the critical set of information that was needed to help users achieve a goal for any given situation. Though this provided relevant information for an instantiated model, we did not have a way to automatically define what information was relevant as filters were provided for either end applications or set situations. Though Rover2 automatically delivered the necessary context for a situation, the ability to handle complex situations needed specific code – context was always the same for situations and varying real-world information as situations unfolded all had to use the same instantiated context. We propose and later explain in detail that a dynamic information ontology provides the ability to encode the logic and provides a dynamic filter to ensure relevant information evolves as situations do. The flexibility of such an ontology and accompanying context management system will allow one to onboard any domain without any coding – the structure and grammar of the taxonomy will allow for a correctly developed framework to do so with no additional coding.

As relevant information is that which helps one resolve a situation, the goal of any context aware system should be to deliver such information. Yet the ontology that can be used as the collection of primitives for that domain can be quite large. A mechanization of the instantiation of the ontology must exist to produce relevant information at any one time. Automatically providing the relevant information allows one to resolve the situation and not have to manage the orchestration of the information structure.

Defining relevance, however, has always been a challenge in existing context aware systems. Relevance is typically manually created per domain [4] [20] [7] while attempts to generalize relevance were done so through the need to create specific filters per situation [14]. Dynamic ontologies provide abstract templates that can be instantiated in one of many ways. The rules of the model should be such that as situations, entities, relationships, and other primitives are instantiated, the model continues to expand and contract as the model's definition dictates it should with evolving information. This results in an instantiated model that is relevant to the current situation at hand, removes the need to further define what is relevant at run time, and is provided with no effort to the end user.

Providing relevant information is not the only area a dynamic information ontology can empower a context aware system. Revisiting how the data was populated in many context aware systems [14], there was nothing to pre-fetch or auto expand the model itself. As we define the grammar for such an ontology, we will show that pre-caching and the expansion of models can also be handled through such a system, thus removing the need for anyone to have to code an end solution for any given domain. The result is a SME can simply create their ontology and the dynamic ontology and context aware system will be the relevant information broker both for data that entered the system as well as data the user hasn't yet asked for, but the system knew they likely would. Ignoring the tenet to pre-fetch data and not working this into an ontological framework, users are forced to create independent software systems to both pre-fetch data and manage such information enrichment rules.

Lastly, goal definition and creation need not be limited to the user of the system. The general use case of an abstract context aware system is one where the user presents a goal or situation to the system, and either through pre-existing data or data that continues to be populated (either manually or automatically), relevant information is presented. However, new information may create the need for new goals to arise – goals that are general in nature and not bound to any situation. Consider the domain of law enforcement and the notion of stolen cars. License plate numbers can be run against state databases that show when a car is listed as stolen or not. The goal to 1) check a license plate and 2) apprehend the current driver of a car are always persistent, yet one need not create a special filter or situation that always checks for this. Rather, the ontology itself should direct one to lookup a license plate ***when the plate information is present***. When that plate is deemed as stolen, the instantiated model should expand and a new goal of apprehending the suspect should be instantiated in the working model by reading the rule built into the ontology itself.

4.3 Dynamic Information Ontology in use

As information evolves for a given situation and is propagated into the instantiated model, the rules in the templated model define what information is relevant. These rules help evolve the instantiated model automatically, whether is be from a consumer evolving the model or any new piece of information being added/removed from the instantiated model. Let us revisit each of the caveats of a dynamic information ontology in section 4.1. Through the use of examples, we will further illustrate each principle.

4.3.1 Encoding of Variable Relationships

When instantiating a model, not all relationships are necessary. Though the model should contain all possible relationships that could exist between two entities, they must not always be instantiated. The model is simply what is allowed to exist. Though that is understood in taxonomies and ontologies today, the deciding factor of when a relationship exists is not encoded into the model itself. Rather, an external system with logic and reasoning of its own dictates when these relationships should and should not exist. We argue this level of variability is just an integral part of the model as the model itself and should not be dictated through independent software.

Consider an ontology about a person. A person might have many attributes about themselves and may be modeled through multiple relationships with other entities. This collection of possible relationships and corresponding entities can be quite extensive. Figure 4.3.1.1 shows all possible relationships and attributes that might exist and be stored in a small template store.

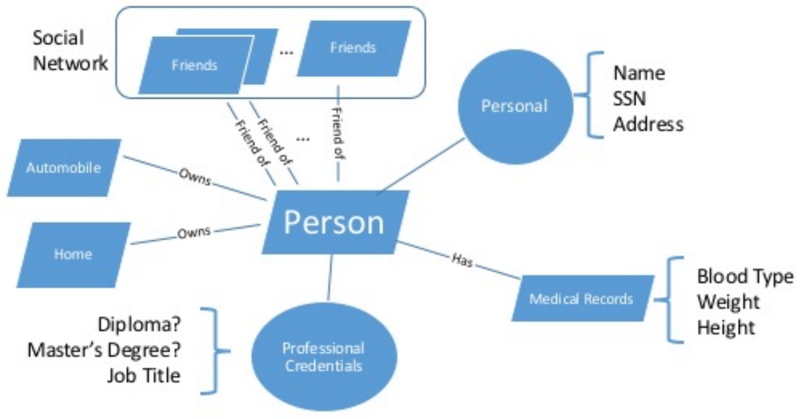


Figure 4.3.1.1 – Sample complete ontology of a person

However, the full model need not be instantiated for each person involved in the accident. Each passenger involved in the accident might really only require the model in figure 4.3.1.2 to be used. However, the template model for **ANY** person (independent of being involved in a car accident situation or not) might be to automatically instantiate the portion of the possible model around medical records., resulting in the model of an injured passenger to look like that of figure 4.3.1.3. The dynamic ontology can be coded to look at the attributes of each person in the instantiated model and if they are considered 'injured', the additional parts of the person template can be instantiated. Furthermore, the dynamic ontology can have an action encoded within itself that could reach out and pre-cache all medical records for the person. For example, we can encode the ontology to direct a context aware system to not only expand the set of information deemed relevant but collect that information from various sources before it is ever asked for. This would be done in anticipation of it being needed given the current state of the situation and all information known to date.

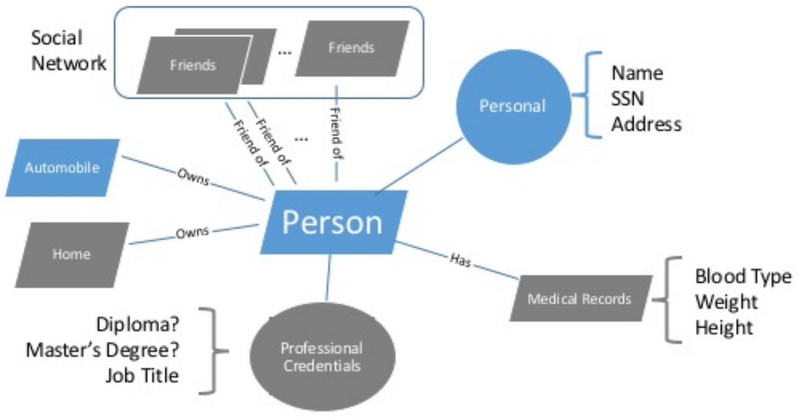


Figure 4.3.1.2 – Instantiated model of Passenger (Refined Person Model)

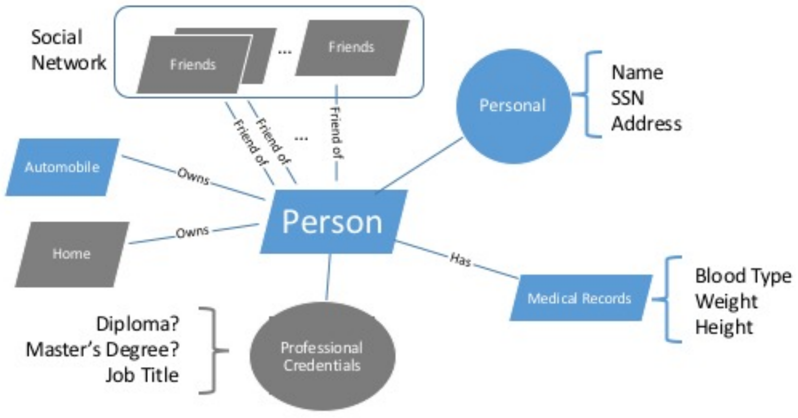


Figure 4.3.1.3 - Expanded instantiated model of passenger as information shows the individual is injured.

We propose that the model of any entity could take this variability of relationships into account. Such a framework that works with such models could automatically expand this model accordingly and ensure the relevant model is always instantiated.

4.3.2 Variable Hierarchal Entities

When we consider how people think about objects, one must be aware of the varying levels or refinement that take place. As a person walks by a building, their mental capacity about the building might only necessitate that they consider the outer boundaries of the building; that is the exterior walls that they can see or are aware of. There is a height the building has, a color, collection of shapes, number of windows, and a number of floors that the person may or may not be aware of. Entrances are present and there might be a glimpse of what is inside the building if windows are present that allow for a view into the structure. If the person has an appointment in the building and they have been inside the building, their mental model is more refined. They know there is at least one space (likely a room) of which they know how to access from within the building. Any stairs, hallways or elevators needed to access that space are known as well as a general layout.

The building maintenance worker, however, likely has an even more refined mental model of the building. All floor layouts (including those that might be under construction and only have planned walls) would be known and a mental model would exist for that worker. Visible items, such as doorways, room numbers, and stairs as well as items not seen by many such as plumbing lines and electrical wiring would all be present. The maintenance worker might not always think about these things when the notion of the building is brought to the forefront of his mind, but the model he has in his head can go as deep as needed.

Just as relationships can be variable as noted in the previous section, the composition of entities can follow a similar path and allow for the auto-refinement or auto-simplification as needed. A rather extensive model can allow for both general and specific representations of the real-world thing they are referencing within the computer system. Returning to the building example, we can create a model of an abstract building. This abstract model might include the relationship ‘contains’ ‘a floor’ relationship, entity tuple. The building itself might have the attribute ‘number of floors’. We could then specify that the number of the ‘contains’ ‘a floor’ tuple is instantiated for the number of times there are floors in the building.

The extent for which a floor is initially modeled can be encoded as well. If a floor is considered ‘complete’ it might contain rooms, access methods, and attributes like ‘number of rooms’. If the floor is ‘under construction’, many child entities simply might now apply. Rather other entities might relate to the room. For example, ‘builds’ and ‘contractor X’ might be associated with the particular floor if this is a high rise. This process of how to extend any particular component of the model can all be self-contained within the model. As the framework consumes information, the entity itself is refined and expanded as needed. Like most of the context aware system design tenets, not adopting this hierarchal context modeling tenet requires consumers to craft both rules around primitive refinement/generalization and software systems to manage this transformation at run-time.

4.3.3 Collection of Actions Goal-Driven Entities Can Take

As we argue that a context aware system’s purpose is to provide information to end users or systems with particular goals in mind, the framework should also have the ability to track and help achieve goals. After all, if there is nothing a user of an information system intends to do with the data, then there is no such thing as relevant data. The notion of the data being relevant means someone needs one subset of data over another.

These goals can vary. The assumed goal could always be simply 'provide what most likely is relevant to the situation to make a more informed decision'. However, at times it is likely that the creator of a system knows when certain information requires some of the entities to take action. The activity required to achieve these goals is nothing more than a collection of goals that change state for entities as well as the fetching of relevant information to assist in this accomplishment. The various entity states can be tracked within the system. As attributes change or relationships are formed with entities, the activity tracker will advance and continue to monitor for proper states.

Revisiting a two-car accident scenario within the domain of a public safety responder, if one individual is hurt on the scene, the goal of 'transport hurt individuals to a hospital' could very well be encoded into the template model by creating an activity named the same. Figure 4.3.3.1 shows a sample activity. Note how once the state of the person is 'injured', the activity is triggered and the system looks for resources (ambulances), performs an external action (calls and dispatches an ambulance), then tracks the various states of entities involved in the activity.

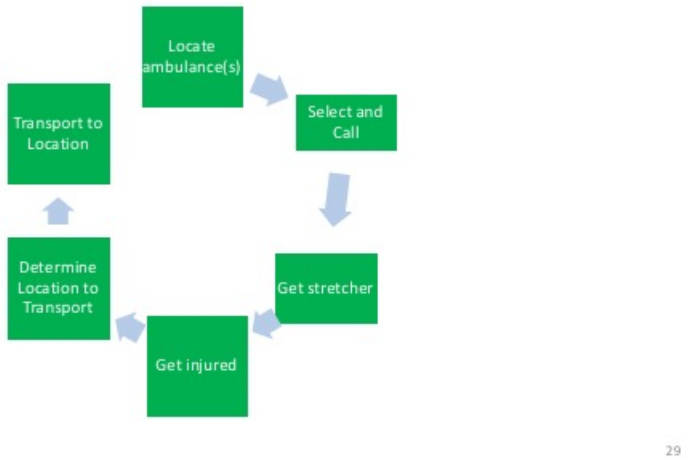


Figure 4.3.3.1 – Sample Activity of Transporting Injured Passenger

This activity would ultimately end in the hurt individual being in a state of 'in a hospital'. Prior to that state being achieved, the framework can do the following:

- Identify what other entities are needed and considered relevant to the situation. For example, an ambulance will need to be identified, a local hospital with space will need to be identified, and the paramedics will need to remove the individual from the wreck to the ambulance.
- The activity can track the status of each step. The 'goal' state can be the establishment of a set of relationships for entities (ex: an injured person can be 'loaded into a' (relationship) 'responding ambulance' (entity)) or simply contain the proper attributes (ex: person: (heart-rate: '<120 bpm'))
- Accomplished via an activity tracker. An activity template needs to be created within the model. The activity tracker will track all relevant entities and relationships.

Activities are not necessary for any model; they are to be used at the discretion of the SME creating the model. However, revisiting the HHC tenant of what a context aware system should deliver, the ability to define *how to work within your environment* should be considered just as importantly as what information is relevant. We claim the modeling that takes place within a domain and various situations can be extended to involve the desired manipulation of data.

As with other primitives, goals need to be hierarchal as well. Designers of context aware systems should allow users to reference and model both high-level or very detailed level goals. Just as goals might have sub-goals, any one goal can be more specific than a related general goal or more general than a related specific goal. Coming back to the car accident again, "tow away the broken-down cars" and "drive off dented car" are both more specific goals of "remove accident-involved cars". In a similar fashion, "clean up debris from the road" is a more general goal than "cleanup dangerous chemical spills".

The need for these hierarchal goals is a result of not always knowing what the exact situation will be for any general circumstance. As information becomes present and situations become more specific our relevance model is refined and perfected; and this should include the goals. This goal refinement is both a by-product of refinement and a necessary method of model creation. As we initially instantiate a general situation we have general goals. More specific situations yield more specific goals, goals that help rectify the baseline in question. The modeling of the situation is what allows us to instantiate more specific goals as the situation unfolds. This is only possible through the required act of building a model that accurately provides relevant goals to relevant information – information that is used to achieve the goal. For example, we would not want to consider the goal of "cleanup dangerous chemical" if there are no dangerous chemicals at the scene of the accident. We may initially just want the "cleanup debris" goal for any accident that has debris in the road. As the details about the debris are made known we can then make the goal more specific.

Similarly, as we do not always know what goals need to be met for a given situation (until our system suggests to the end user that is), we may not know what actions are required to reach the goals before us. Just as entity / attribute combinations can dictate what relationships or entities are considered relevant to any one situation or model, having a goal can make related primitives required. As goals can be met through pre-determined actions, goals can trigger the necessary existence of activities. For example, if you are dehydrated and the goal is to become hydrated, there is a required activity of "increase hydration in the body". Goals are not limited to simply instantiating activities, supporting entities that support the ability to perform the action can also be triggered by goals. Back to the hydration example, water of some other liquid would be a require entity that, prior to needing to accomplish that goal, was not considered a relevant entity. Such a goal could make this entity relevant and instantiate it. The triggered primitives can be specific or general as well and vary as a goal is refined or broadened throughout its hierarchy.

5 Introducing ADIM as a Means to Encode Relevance

In this chapter I introduce the Automatic and Dynamic Information Model methodology (ADIM) as a new approach to defining dynamic ontologies. ADIM is a unique and novel way to think about ontologies. Using ADIM, ontology authors are not bound to the normal static nature of modeling and presented with a dynamic modeling methodology. This methodology is intended to provide relevant information for any situation based on existing, instantiated models and information. ADIM models, when instantiated present all information it deems relevant for a given situation. As situations evolve over time, ADIM provides the ability to automatically expand models at run time in effort to continually evolve the relevant information as needed.

This self-evolving trait of ADIM satisfies the need for the manual context management tenet. I enable this use of template auto-evolution by implementing them within a graph database, allowing for exploration and expansion as needed or warranted. The resulting methodology is one which delivers the notion of the dynamic information ontology and enables a novel modeling approach.

I will briefly discuss ADIM, during which I will use Cypher [49] notation as I describe examples of ADIM in use. I will explain later in chapter 6 why Cypher is used with the implementation of Rover3 is explained, but for now focus on ADIM.

5.1 ADIM

The argument for ADIM is that one creates an ontology that allows for the flexibility of the types of models we have described above. Reviewing typical ontologies like RDF [50] and OWL [51] [44] [52] we hardly find the level of flexibility we believe one should have. The notion of a dynamic ontology has been proposed in [53] and [54], but those concepts were targeted at automatically updating a static ontology and removing the need for one to keep the ontology up to date. Components in these existing ontology frameworks are binary in nature; they either exist or they don't. Despite allowing for any ontology to be defined, systems that tend to use ontologies [8] [20] [48] [55] are limited to the structure of the ontology itself. The ontology must be fully instantiated, or a specific filter must be created for each use case, before the use case is exercised. Furthermore, the rigid nature of the taxonomy only allows for some entities to exist or not; there just hasn't been an ontology that encodes the variability of the domain itself.

ADIM changes that. Rather than needing an external software system for every consuming system of an ontology, ADIM (and a system with the same properties as Rover3) allows one to simply define an ontology and only consume what is relevant. Authors of the ontology can encode many different relevance models within the entire ontology itself. Using the ADIM grammar, one can dictate which relationships, attributes, subsets of the ontology, actions (the consumer of the ontology should take), or pre-caching should be instantiated at run-time throughout complete models. The result is only a subset or pruned version of the overall ontology is instantiated at one time or expanded or contracted as the instantiated model evolves. This reduces the need to create a filter for each iteration of an ontology and is a more complete model of any domain.

Recall that a situation is what defines the context, or the subset of the entire possible collection of information of a surrounding environment. As such, an ontological designer can encode the relevance for any number of situations. This is done by enabling or disabling sections of the model using ADIM grammar. Working through the entire possible model for any domain or set of domains, the creator can carefully cauterize sections that do not assist in the remediation of the situation. Situations alone need not be the initiating characteristic for any pruning/expanding function – entity/attribute tuples and other primitives that may exist in the instantiated model and dictate how to expand/prune a large model. The ADIM grammar is designed to allow for this as well. The application of ADIM grammar in an ontology is broad – it can define variable relationships, sub sets of entities to instantiate, prune attributes, define actions or goals to take, and even work to deliver information pre-caching.

Adding the ADIM verbs and actions to an ontology dictates to Rover3 (or any system that would instantiate the ontology) how to prune and produce only the relevant sub-graph. Thus, the goal of ontology creation or modification of an existing one is to understand the ADIM grammar and how to apply it as they create a new or enhance an existing ontology. As the ontology can always be updated, future iterations or additional situations can easily be added to an ontological model.

In creating a specific ontology within ADIM, one should work to ensure the resulting instantiated models are relevant. The ADIM grammar is a tool that allows an ontological model to always produce a relevant model, but it does not guarantee the resulting instantiation will be relevant. Like any programming or scripting language, the resulting action the ADIM grammar will drive could result in an instantiated model that is difficult to manage or more vast than it should be. While tools cannot always prevent this (an example is a fork bomb that can easily be created through a scripting language) one needs to provide guardrails around such performance issues. Aside from proper modeling techniques, the context aware system must optimize the management of the relevant data set. As context aware systems need to be suggestive in nature, anything that presents the instantiated model must ensure the end user can remove portions that are deemed irrelevant. Additionally, subsets of the ontology that were not instantiated should be identified as relevant at the direction of the end user. For example, if a model dictates that health records of the 300 subjects are not relevant but the weather at the given moment it, the system instantiating the dynamic ontologies must allow a user to prune unnecessary components while adding others.

Not all components of an ontology need to have relevance encoded within them. Anything that does not have a relevance function used to instantiate it is considered default and will always be considered relevant. Likewise, using some of the other features of ADIM, one can assign hierarchal rules or fuzzy matching to the ontology, removing the need to have to discretely define how every single part of an ontology could be instantiated for a given situation.

5.2 ADIM Grammar

The full grammar ontology is outlined in chapter 7 but we will briefly explore the high-level features of the ADIM grammar. The ontology and ADIM grammar together are a superset of the Cypher language (the decision to use Cypher is explained later but a result of the data base chosen to model and store ontologies). A standard, non-dynamic ontology can simply be written in Cypher for Rover3 to instantiate. Without any ADIM rules, the entire ontology would be instantiated as situations or entities are to be created in the Context Store (more to come in chapter 6). Using ADIM grammar and revisiting the traits of a dynamic ontology as outlined in section 4.1, authors can ensure the following actions take place on the instantiation of an ontology:

- **Variable Relationship** – Any linking of two entities can be a function of a specific situation, a collection of entities and attributes, goal, or active activity in the instantiated space. The author can cauterize or enable such relationships with the ADIM grammar accordingly.
- **Enriching Data** – Through the existence of attributes for any one entity, an author can indicate within the ontology data sources or other URI actions to fire. The result can place additional or enriched information into the instantiated model.
- **Goals and Actions** – By creating a situation or evolving the collection of information in the instantiated model, goals and actions can be considered relevant. Authors can encode the ontology to trigger an action or goal based on the existence of a situation or a specific entity/attribute tuple. Likewise, goals or actions can instantiate additional entities as the ADIM grammar permits. Lastly, an author can ensure that a situation being instantiated created new actions or goals when determining the relevant portions of an ontological model.
- **Dynamic Situations** – Not all situations lifecycles need to be defined in the ontology – that is, the ontology can define when new situations are to be created or existing ones should evolve. Authors can allow for goals or entity/attribute tuples to spur new situations, resulting in infinite ways situations can be stitched together.

An example of the grammar can be found in automatic pre-fetching of enrichment data. Consider again the example of a vehicle where we know the license plate. A simple grammar statement that would fire an action based on a value would be:

"Action GET http://website.com submittedVariable pointerVariable"

This means fire the action of querying the web service with the submitted variable and place the return value in the pointerVariable. Using this grammar as we craft the ontology, a specific entry within the ontology could be:

```
CREATE (m:CarOntologyroot:Car:car:entity {licensePlate: '', isStolen: '', preCacheTag: 'Action: GET http://localhost:8080/Rover3/tagQueryDatabase m.licensePlate m.isStolen'})
```

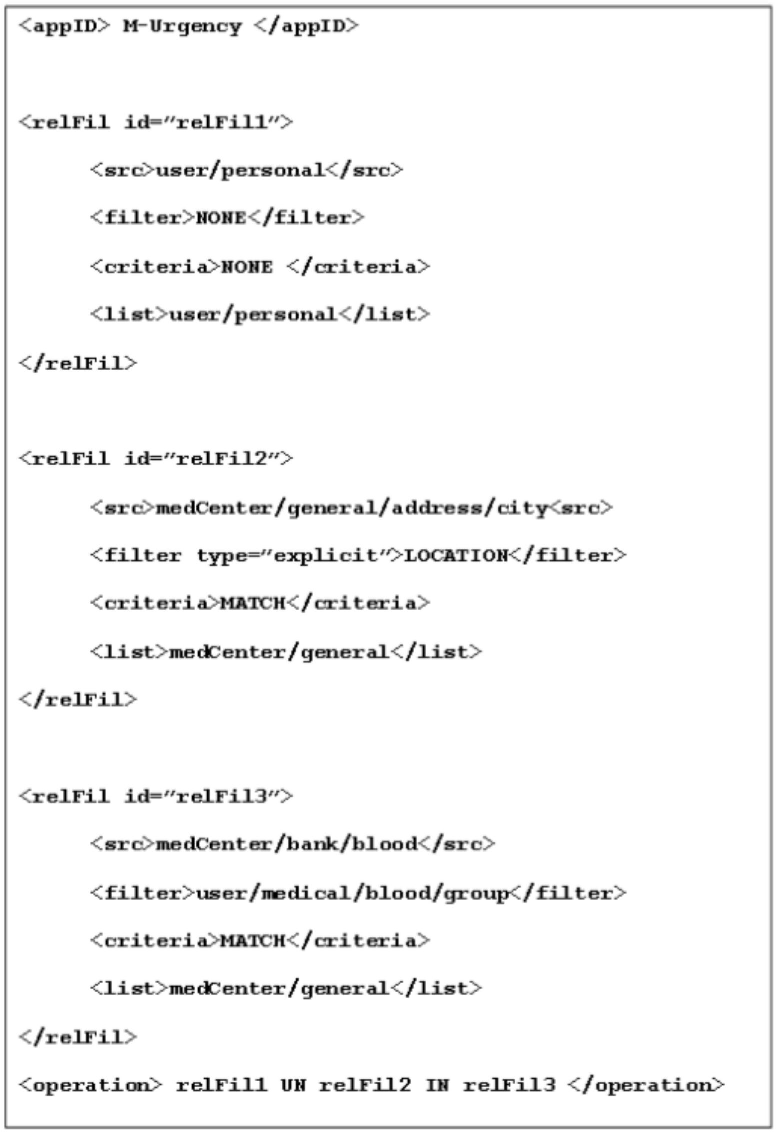
Spoken in a natural language, this says if the license plate of the vehicle is known, tell the activity manager to go query an external web database.

ADIM ontologies can be re-used and extended, allowing for models to span multiple domains or use cases. It is through this grammar that we can encode relevance. Associating relationships and the instantiation of entities through discrete rules or fuzzy logic, we provide a way for the ontology to define what is to be instantiated into the context store at any given time.

5.3 Advantages of ADIM

Existing systems employ the RoCoMo methodology in order to define and present context in standardized methods. Doing so, authors of ontologies and consumers of context are ensured the information they are consuming will adhere to specific rules about the existential structuring of said information. This information, however, is static in nature and does not vary at run-time. ADIM, when used with RoCoMo ontology modeling definitions, continues to deliver this same information structuring assurance. The benefit of using ADIM is in the modification of relevance models at run time – we still are delivered relevant information for a given situation but the effort to evolve what is relevant over time is drastically lowered as Rover3 can do this automatically as it executes the ADIM logic built into the ontology. Reflecting back to our tenets of a context aware system, "Remove the need for manual context management" and "Proactively fetch information and expand models" were two such tenets that one should use when building such systems. ADIM helps delivers on these tenets.

Rover2 is an example where these tenets are not built into the system or modeling framework. The result is system designers need to account for this in unique ways each time they encounter a new domain or application that will consume context provided by Rover2. Context on a per application or situation basis is delivered by creating filters for each discrete scenario. While filters can be re-used, the instantiation of such information still requires external software. The result is a user of Rover2 knows what the generic situation may look like for anyone situation, but as situations blend and evolve, the consuming systems must have logic external to the context aware system that manages what is instantiated, how to blend it, etc. Figure 5.3.1 shows an example of the filter that helps define relative context. Outside of the filters, there is nothing else to define what is instantiated or how situations should be merged or managed.



```
<appID> M-Urgency </appID>


<relFil id=”relFil1”>
      <src>user/personal</src>
      <filter>NONE</filter>
      <criteria>NONE </criteria>
      <list>user/personal</list>
</relFil>


<relFil id=”relFil2”>
      <src>medCenter/general/address/city<src>
      <filter type=”explicit”>LOCATION</filter>
      <criteria>MATCH</criteria>
      <list>medCenter/general</list>
</relFil>


<relFil id=”relFil3”>
      <src>medCenter/bank/blood</src>
      <filter>user/medical/blood/group</filter>
      <criteria>MATCH</criteria>
      <list>medCenter/general</list>
</relFil>
<operation> relFil1 UN relFil2 IN relFil3 </operation>
```

Figure 5.3.1 – Example context filter used in the Rover2 system.

Proactively fetching information for instantiated models is not something that exists in generic context aware systems. If one were to ignore this design tenet, external software would need to be written for each domain or model. Ensuring this feature exists in the

ADIM language allows consumers to benefit from data-enrichment by simply writing a rule and no longer requiring additional software for this workflow.

5.4 Modeling More than just Surroundings, Modeling Thought

Computer systems that employ and use these ontologies make the decision when certain relationships are considered relevant. The software receiving the information itself tends to have logic that defines when one relationship is more relevant than the other. Yet in ADIM we allow for these relationships to be a part of the model itself. Either through existing relationships, entities, or attributes, we can encode that variability as a function of the aforementioned within the ontology. This aims to capture the 'thought' process that mirrors the mental model we humans might employ and place all of the login within the model itself and not distributed across the data store and an external logical reasoning system.

As noted in [14], the use of filters allows systems to define what is relevant. While imperative to a context aware system, the filters themselves (including next steps, goals, and model refinement) need management as the situation itself unfolds. For example, let's revisit the need to gather the social network of the suspected terrorists as outlined in chapter 2. If you had to gather the social network of those individuals manually and you are only armed with a computer and nothing more, you might simply start your search in google. A quick search of both names yields they each have a Facebook profile. So, by default a next step might be to search each user's page, gather what public information exists, noting other social ties they have, where they have lived, etc.

Yet when you look at the 2nd google search result you note something different. The result for person A is a link for a running race with results. The result for person B is an academic paper. This delta in the type of results yield a different flavor to the same action you would take for each. You now have some context about the types of social networks they are contained in. Person A is likely connected with other runners and athletes while person B likely spends some of their time with other academics. The results of their efforts are different as well. Person A may have run multiple races, so the goal for the refined search of 'what have they done' would be to look for races for person A and look for papers for person B.

All that really differs is where you perform your search and what your goal is. You are still going to look for 'something' related to the individual. You might do a more refined google search such as 'Person A's name + race results' for the one and 'Person B's name + research paper' for the other. If you knew something about both domains you might go directly to other database sources. You might go to athlinks.com and search for Person A as that website might have the most comprehensive list of race results. If the person has run for university teams, you now know their social network likely extends to that which represents a NCAA athlete. You can then check the school they attended and gain a more refined model of that person. Similarly, you could go to google.schaolar.com and search for Person B. Here you have both results of other papers they may have written but also have a new social network, namely their peers with whom they have published papers with. University affiliations again might arise and the model of person B is equally as refined as it was for person A.

In both scenarios, we started with the general model of a person. We expected google to return a result and we explored social networks with the Facebook results. We then did a more refined search given the results that returned but leveraged different repositories in order to do so. In each situation, our model for search was the same, we just simply took the information that was given to us and refined our approach. While we could have used filters to get the final product, the decision-making process would have had to have been uniform for both person A and person B. As the filter is static, something would have had to have told us that we need one filter over the other. Rather than a filter, if the ontology itself defines what information is relevant and where we should search, the need of a filter is removed. The result is expansion of the model at run-time given the data at hand. Such data expansion and refinement is very general in nature and doesn't need special code. It is the actual model that drive the software, not the software driving which models we need to use.

This model we believe more closely resembles what we consider. Our claim is that we can encode this dynamic model of relevance into an ontology itself. Using the grammar defined within ADIM, we argue that thought, though requiring a bit of work through the creation of the model, can just as easily be modeled as any other domain.

6 System Overview

In this chapter, I layout the design of Rover3 – a context aware framework that implements the tenants outlined in chapter 3 and the modeling features in chapter 4. Rover3 acts as a middleware system for both end users and applications. This system provides a domain agnostic platform for delivering and exploring relevant data. Applications and users that need to store, retrieve, and model data are able to do so through simple API calls. Using RESTful APIs, any application can simply request, create, store, and explore data for a particular use case. The goal of Rover3 is to reduce the need for customized software when the need for relevant information is needed. Such a platform re-defines the notion of an information taxonomy as this is the driving force behind data manipulation, extraction and refinement.

6.1 Rover3 General Framework

The architecture put forth in [14] was one that we believe is appropriate for a context aware system. The approach in Rover2 provides context for any situation. Rather than be bound to mobile computing, location based awareness, or any typical context delivery system, the authors of Rover2 laid out a design that allowed for generic context to be delivered through a common middleware platform rather than a custom one so frequently designed.

Though similar in nature, we believe modifications to the existing Rover2 architecture is necessary. A visual of the architecture is found in figure 6.1.1 below.

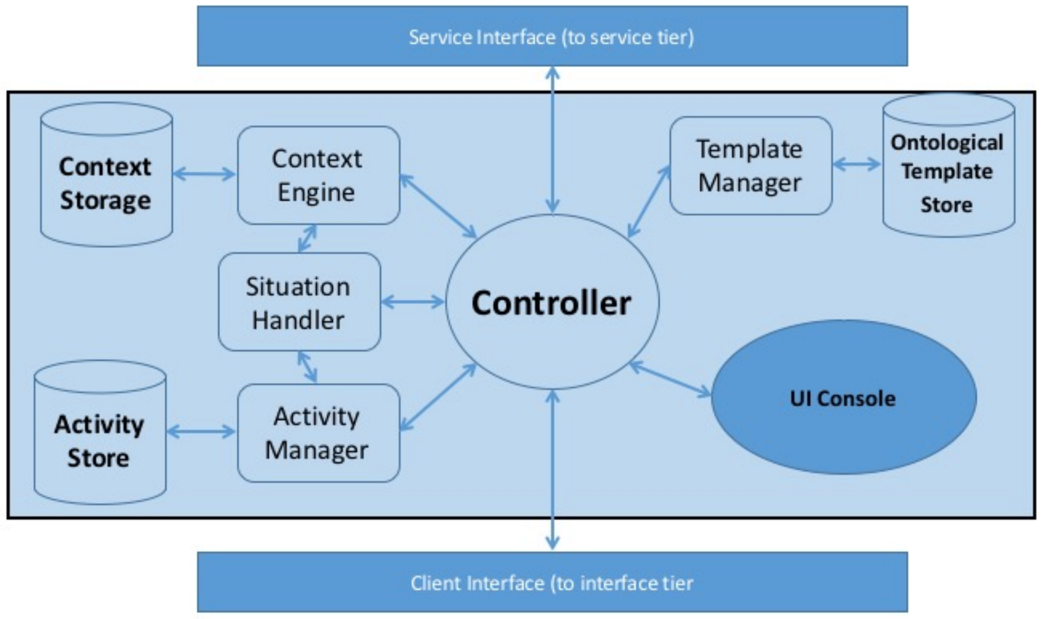


Figure 6.1.1 – Rover3 System Architecture

Defining the main parts of design the system:

- **Controller** – Dictates the actions each other module is to take. All information flows through the controller. This most resembles an HTTP server. Heartbeats are sent when context is updated from the controller to each module in an effort to keep context relevant while allowing modules to run autonomously. These heartbeats help evolve the instantiated or relevant information.
- **Template Manager** – Stored the generic models that the system works with. The Ontological Template Store is the collection of models that SME's generate for the system. The Template Manager interfaces between the store and the controller. Requests for models and expansions are handled here.

- **Context Engine** – Stores the real-time, instantiated models. Having external inputs relayed through the controller, the context engine helps evolve the relevant information as it executes the ADIM logic. This could also be considered the active cache or database for the active portion of the system. Relevant context is delivered via the context engine. Though the controller is the module that directly delivers data to the end user or system, it is the context engine that defines what is to be delivered. The Context Storage is the actual repository of relevant information. This is the collection of all instantiated models. It may contain only active or active and existing models. These are discrete models with specific data and instantiations unlike the template store which contains abstract and vast models.
- **Activity Manager** – The activity manager monitors activities that the context engine deems are necessary for given instantiated situations. The Activity Store is nothing more than a logical partition of the ontological template store. As activities are instantiated, the activity manager track status of entities and help drive outside users and systems to complete the goals the activity seeks to accomplish.
- **Situation Handler** – The situation handler manages individual situations through the full life cycle. As situations appear and wain, the situation handler monitors the situation, looking for opportunities to merge multiple situations together as well as maintain the state of situations.

- **UI Console** – This UI console is the mechanism that allows users to view the current state of the situation graph. Users can select which graph they wish to view as the system can work with multiple situations at a given time. Data and its relationships are visualized through this UI. Users can expand and shrink both instantiated models at the given times as well as proposed models. Attributes and their values are shown as well. Logging for each engine is displayed through a text console. As data is entered and processed, the action each module takes is logged. This console is an effort to help users understand how to use the system and better showcase the prototype. It will also serve as an interface for users to automatically expand models as they see fit (an extension of proposed API calls to be discussed later).

Rover3 is designed to be used by users or applications alike. As an information management and brokering system, Rover3 caters to an end user or an application. Designed to act as a real-time or long-term data-store, Rover3 is as little or as much of a database as needed for any application. Through the tracking of various situations, Rover3 provides relevant information for each situation. Applications and users can access this information through the use of web service calls or visual inspections through the use of the UI. As the information brokered by Rover3 is to be used to help accomplish goals and help influence the decisions the UI and web calls should allow for modification of the instantiated model. Rover3 is not meant to be authoritative but rather helpful.

6.1.1 Graph Database as the Data Store

The data stores for each of these are handled through graph databases. Graph databases allow for unrestrained modeling and were chosen based on the ‘Extensible modeling framework’ tenet. Existing database technologies such as SQL or even NonSQL solutions require either arduous schema updates on a per-model scenario or relationship management overhead. The ability to model entities and relationships with graph databases affords one the flexibility to model any entity, relationship, or situation. Activities can be modeled through the collection of entities and desired attributes. The limits of how to model are merely in the minds of the SME’s – designers of the models. There is no limit on the number of relationships to relate independent entities or decompose high-level entities into more detailed explanations. The unbounded and unstructured nature of the graph components as well as the attribute stores (similar to JSON objects) allows for simplification of data management and better scalability.

We argue this graph database is better for data and ontologies. As we argued above, computer systems and models should strive to minimize the abstraction of the reality around it. Convention SQL databases, though efficient in indexing, require a bit of massaging of information through code to convert stored data into processed and delivered data. The standard MVC approach to software design requires an extraction of data in one format, massaging, merging, and processing, only to deliver in another. Using graph databases, data is considered to be in the final state. The need to massage is removed by leaving the data modeled as the domain or situation deems relevant. Processing this data (aside from the functions of Rover3) are left up to the end user, but the removal of the need to massage the data outside of the data store frees the system of restrained representations and minimizes the effect of data representation.

We now take a deeper dive into each of the other modules, explaining their functionality a bit more while detailed some of the specific calls and rules of engagement with one another.

Type of Call	Example	Explanation
Create Situation	http://RoverServer/createContext /createEvent/carAccident	A car accident situation is created. All necessary entities for this graph are instantiated. A situation ID is returned.
Query model composition	http://RoverServer/getContext/situation/carAccident/061705	The situation model is returned as it exists at this point for ID 061705.
Populate Data	http://RoverServer/createContext/updateEvent /carAccident?person2=true&first name=John &last name=Doe&address=123FakeStreet &injured=no&id=061705	Situation ID 061705 has some of the data updated for the corresponding models
Query specific entity	http://RoverServer/getContext /carAccident/061705/person2	Returns all data and structure of the person entity for the given accident; more specifically person 2.

Table 6.1.2.1 Sample Controller API Calls

6.1.2 Controller

The controller is the main interface within the system. All communications from one engine to the other goes through the controller. One main reason for doing this is to alert the entire system that context has been updated. Every time data is entered or processed or models are expanded or instantiated, that is a relevant change to the context at hand. Heartbeats are sent out across the system via the controller. Both the context engine and the activity manager have the ability to take actions based on data at any given point. All communication is handled internally within the system and within the programming language the system is written in. Sample calls are presented in table 6.1.2.1.

The controller additionally serves as the external interface between the Rover3 system and the end system or user. This external interface serves to deliver context, describe context (and models), accept new information, define new situations, and end situations/events. Using a RESTful API, commands are sent to the controller for such queries and updates. Both meta data about the relevant information as well as specific information can be retrieved through these calls. As any receiving entity does not know what is necessarily relevant (Rover3 helps deliver just what is relevant), nor do they know what the actual data is that exists for the situation, the controller allows for all of that discovery to take place. By querying for the model composition, end users or systems can see what the actual structure of relevant information is. Entities and relationships are labeled. When the actual data itself is required, specific primitives can be returned, as well as whole or sub graphs.

Data returned from the controller to an external entity is in the form of a JSON object. As relevance can vary from second to second, the structure of the data is just as dynamic as the actual data values. JSON objects allow for proper key:value store/organization of the data and provide an unstructured, yet detailed mapping for any data structure. This JSON formatting easily captures the state of the situation graph and any subgraphs. As entities can be quite large and expansive and contain many attributes, a JSON object allows for easy organization and indexing of any entity.

6.1.3 Template Manager

All of the models and logic that Rover3 can instantiate a model are stored in the template store. The template manager is the interface to this store. As the activity manager or context engine need models delivered, the template manager does so.

The ontological template store is merely a collection of models within a graph database. Similar to other ontologies, the template store is a collection of abstract entities and relationships. Entities need to be modeled with all attributes they might have directly assigned to them. Logic is encoded in this store within the ontology using ADIM.

Relationships link other entity types together and references can be made to more general entities. For example, an ontology for a person can be rather vast and contain hundreds of attributes and another hundred or so relationships. As a domain may contain many other situations or events that contain one or more persons, the model must reference the general entity of person. This requires us to label our primitives. The result is we simply walk the labels to refine to generalize any given primitive.

The database of choice for Rover3 is Neo4J. Neo4J was chosen as it was one of the more mature graph databases. The query langue Cypher [49] is what Neo4J uses to create, query and mutate entries within the database. Templates are merely a collection of multiple queries that define all primitives as well as possible relationships between all possible entities. As the template is a collection of individual queries, labeling is used to decipher between entity definition and references to the entity. Examples of such models can be found in figure 6.1.3.1.

```
// Person Entity
CREATE (person:Entity:Person {first_name:'', last_name:''}

// Person instantiation
CREATE (person:Person {first_name:'John',
last_name:'Smith'})"});

// Referencing Person in another component of model
Match (auto:Automobile), (person:Person)
CREATE (person)-[:OWNS]->(auto);
CREATE (person)-[:DRIVES]->(auto);
CREATE (person)-[:RIDES_IN]->(auto);


//FOAF, with new person type
Match (person:Person)CREATE (person)-
[:HAS_BROTHER]->(brother:Person)CREATE (person)-
[:IS_RELATED]->(brother)CREATE (person)<-
[:IS_RELATED]-(brother)CREATE (brother)-
[:IS_BROTHER_OF]->(person)w entity types for person
```

Figure 6.1.3.1 - Sample Model written in Cypher

The template store is the aggregate of all possible ways to instantiated something, such as a person entity in chapter 2. Yet all possible options are not always used. The storing of the model within a dynamic ontology allows only the relevant components of that template to ever be instantiated. Templates are written with ADIM and are re-usable/extensible.

6.1.4 Context Engine

As data is populated with the system and models are instantiated, the context engine manages that information. Its purpose is to contain the relevant context for any given situation (or situations) and broker that data back to the end user or system via the controller, satisfying the design tenet to do just so. Recall, that context can be known information for that given situation or placeholders for information that is relevant and needs to be discovered. Through its own instance of Neo4J, the context engine stores all instantiated data in that graph database instance. As templates are used or referenced, the context engine replicates those components into its own instance. Using graph traversal and copying components within the Neo4J libraries, the subcomponents of the ontology are easily referenced and copied. One is presented with a situation graph for each situation that is being managed. (Note: throughout the rest of this dissertation, situation graph will be used synonymous with the concept of the given situation being stored in the context store).

The context engine maintains the context store where it stores both the relevant, filtered models and all relevant information for any given set of situations. As situations are created by external users, the context store is first populated with a filtered, or relevant version of the ontology from the template store. Only the portions of the ontology that are considered relevant are instantiated within the context store. As external users populate the context store (and the controller sends updates via hearbeats) with specific information about the situation, further portions of the ontology can also be instantiated and placed into the context store if the logic in the ontological store deems it necessary.

As the context store stores both relevant filter versions of a larger ontology and actual information for a situation, the controller can return either the relevant information structure of the actual relevant information itself as dictated by the service call. As both templates and instantiated data stores are all graphs within the graph database, subgraphs or entire graphs can be returned as well and individual entities.

6.1.5 Activity Manager

The activity manager, among other things, ensures the relevant information is always up to date, either through the expansion of models or pre-fetching of relevant data. As heartbeats are sent, the activity manager looks to see what rules are or are not in place within the context store. Any rules that define activities to fire do so if the combination of context and ontology rules define so.

The activity manager can also process larger, templatized activities. As mentioned earlier, goals are a part of relevant information. A chain of entities and desired entity states can be modeled and instantiated to help achieve that goal. For example, consider the two-car accident again. If a passenger is noticed to be injured, the goal might be to transport the injured to a safe place such as a hospital. The activity manager can expand the situation graph to include paramedics and an ambulance. The location of the passenger would want to change from ‘in-vehicle’, to ‘in-ambulance’, to ‘in-hospital’. During these state changes the activity manager can be fetching information about local hospitals, seeing which ones have room for a patient as well as which ones might not be able to treat a patient with the types of injuries they are suffering from.

The activity manager does not necessarily accomplish the above activities, but merely tracks the state changes, though it has the ability to do so. As state changes are tracked, relevant information is adjusted and can be brokered to end users. The actions that are coded into the model can be performed by the activity manager and could actually reach out to external entities. The framework of Rover3 does allow for external messaging. The calling of the ambulance, the reserving of the hospital bed, and transferring of medical records are all possible actions the activity manager can undertake provided those actions are RESTful actions.

6.1.6 Situation Handler

Rather than have external software to manage the various states a situation can go through, the situation handler is always monitoring what situations are instantiated. The goal of this monitoring agent is to take any and all possible evolving actions on the situations in the situation graph. These actions include refining or broadening situations, creating new ones, removing situations that are no longer deemed relevant, or archiving completed situations.

Using the hierarchal properties of primitives in ADIM, we can enact actions on the situations according to the rules in the template store. As information is added to the situation graph and heartbeats are sent via the controller, the situation handler can refine or broaden situations. Rules for more specific situations can fire a trigger to instantiate a more specific situation in the context store, thereby also instantiation relevant primitives to support that situation. Both the supporting general situation would continue to exist with the child situation. The same chain of events can remove the specific situation if deemed necessary.

Using outside information to fill out the situation graph, the situation hander can also create situations. Just as above, the handler will continuously monitor all instantiated primitives that have situation creating rules, instantiating if certain conditions materialize.

An example of this would be a person entity that is deemed to be hurt really bad. The enriching of the person entity might fire a trigger to create a injuredPerson situation that is separate or combined with the existing situation.

Lastly, as activities complete and goals are met, the situation handler marks a situation as no longer active and archives them. Doing so does keep entities and other primitives open for discovery as they represent previous representations of real-world primitives.

6.1.7 UI Console

The UI console is merely a visual look into the current situational graph. This module serves to allow end users to see what data has been instantiated, what values exist, as well as templates that have yet to be expanded. Tooling will be engaged to show what next-level components of a model exist for any given entity.

The tool will also look to allow end users to populate data and expand models as the user feels fit. As Rover3 is not designed to be the authoritative source on context and actions but rather a suggestive source to help influence users make decisions about how to resolve a situation, the system must allow for such input and manipulation of relevant information. As one of the controller functions will be 'expand context', users will be able to expand a model, drill into a hierarchal entity, or add relationships within the ontology as they see fit.

A text console will accompany the visual aid as well. This component will articulate all actions the various modules undertake. Heartbeats and resulting triggers will be listed as the system receives and processes data. The logic the system takes based on the ontology being referenced will be expressed, providing a play-by-play as to how the system is ‘thinking’ and behaving.

6.2 Domain Independent Framework

The Rover3 framework is one that is truly domain independent. While individual ontologies will need to be created for any given domain, the modeling of data, the underlying system that manages and presents the data as well as the mutation of data will truly be domain independent. This independence is realized as SMEs can build such a system without any coding. Through the use of the Rover3 and ontology grammar, a SME can define an ontology that not only lays out the structure of entities and the relationships between them for any given domain, but also capture how data and relationships evolve and vary based on individual content.

In order for a SME or organization to utilize Rover3, all they need to do is define the ontology themselves. There is no need for additional software to be written when a new ontology is written as the framework will handle the information processing itself. Instantiation of models and expansion of data is all handled automatically as Rover3 processes the logic encoded in the dynamic ontology.

7 Putting it all Together – Dynamic Ontologies in use via Rover 3 and ADIM

We now turn our attention to the specifics of how one uses Rover3 ADIM to create their own dynamic ontologies and deliver relevance. There are two classes of Rover3 users. The first is a set of users that craft ADIM ontologies for Rover3 to use, instantiate, and model. These people are SME's in their field of work. They understand how the various entities in their domain relate to one another and can add to existing ADIM models in an effort to adequately represent what is necessary for real-time information management. These users are the ones who create the ADIM models in the template store and use the verbs in sections 6.1 – 6.4 below. The second set of users are those that benefit from the by-product of the Rover3. They can view the models on the template store in order to explore how a SME believes a model can expand or exist. As models are instantiated and a relevant model is provided to the context store, they can view and explore what exists.

The following sections act as a reference / API manual as it outlines the full ADIM grammar; rules one can use when defining an abstract relevance model as well as the web calls the controller uses to receive input from external users and systems. Each dynamic feature of dynamic ontologies is outlined below both with a detailed explanation of each way the specific component of the ontology encodes relevance as well as the full grammar or ontological spec. As the Neo4J language of Cypher is used to create the dynamic ontologies within Rover3, the examples are expressed in Cypher. Some of the matching and constraint grammatical rules are simple extensions of the Cypher language. Rover3 is designed to process the ADIM grammar outlined below.

7.1 ADIM Grammar

We start with the general ADIM grammar concepts and common components that make up the ADIM language.

7.1.1 Basic Grammar

The ontologies that are stored in the template store are done so through the use of the Neo4J language Cypher. Stringing together multiple statements that embrace the ADIM relevance grammar allow one to create a dynamic ontological model. All primitives are modeled through the crafting of individual statements. Entities, Situations, and Activity states (to include goals) are all modeled as nodes. Relationships are modeled the same as their Neo4J namesake – the relationship. An activity is really a string of activity states (nodes) and links (relationships).

We use the Neo4J label operator ':' to define or refine the specific primitive being defined. Every node in Neo4J needs a label. As entities are nodes, we must label each entity. The entity labels act as the type of entity. Labels read left to right in terms of generality; that is the left-most label is the most specific type for that primitive and the rightmost of the least specific. Every primitive that is a node (Entity, Situation, Activity State) has at least three labels, with the right-most label being that type of primitive it is. The left most label is the label to be used the immediate stanza and is not used beyond the single line. This label can be anything but is used when referencing that primitive in any other part of the stanza. Relationships do not require a starter node nor do they require a primitive declaration as they are automatically known to be relationships. Activity links do require a general label called *activityLink*.

Labels can be re-used in the template store and should be re-used in order to implement an ontology over and over again. Consider the following two examples:

CREATE (m:PersonOntologyroot:Person:person:entity {first_name:'', last_name:'', id:'', ssn:''})-[:worksAs {situation_contains_text_professional: 6, situation_exists: 'careerQuery'}]->(o:Profession:profession:entity {title:''})-[r:employedAt]->(p:Company:company:entity {name:''})

CREATE (y:Person1:person:entity),(z:Person2:person:entity)

In the first of the above two stanzas we create a person ontology which acts as the general person. All future instantiations of 'person' in the context store will always instantiate that model and take into account the ADIM verbs when expanding relationships or not. The second stanza creates two more 'persons' for other use in the template store. The use of the 'person:entity" labels tells Rover3 to re-use the person ontology. In both of these examples the single character (m, then y) labels are the leftmost labels that are not used but required due to Cypher syntax rules.

This use of multiple labels also doubles as a method for providing hierarchal models. For example, when creating models of situations in the template store, a general situation of an accident can be defined. A subsequent carAccident can also be defined that would inherit properties of the accident model. More specific activities and rules overwrite conflicting rules and triggers of the more generic model.

The following example shows the creation of a situation labeled carAccident (which is also a generic accident), an entity which is Person1 (which is also known as a generic person), and the linking of the two in a car-accident through the relationship involvedIn:

```
MATCH (f:CarAccident:accident:situation
{location:’’}),(g:Person1:person:entity)
MERGE (g)-[:involvedIn]->(f)
```

Note the re-use of the throw-away labels f and g and the Cypher language rules. The person type is called Person1 in this example as the model this stanza was taken from had multiple people in the car accident and the use of the label Person1 allowed Rover3 to differentiate between that person and Person2 (not shown but used in other stanzas with the dynamic ontology this was pulled from).

This dissertation does not go over the varies properties of the Cypher language and one should refer to the documentation of that language for more info. The template store will accept any acceptable Cypher stanza but Rover3 has only been designed and tested to work with the following Cypher keywords/concepts:

CREATE

MATCH

MERGE

SET

RETURN (simply for stanza completion)

attributes

nodes

relationships

7.1.2 Attribute Tests

There are a few places in ADIM where an attribute test is an optional or required part of the language used to determine relevance. We adopt and process a subset of the Cypher language operators in order to allow attribute values to determine when to instantiate a model or modify information in the context store at run time. Attribute tests can validate user defined values on attributes of nodes and relationships. Both text and real numbers can be used. Depending on the verb calling the attribute test the attribute can be a stand-alone attribute or a label-attribute pair, delimited with a period '.'.

These tests are valid tests in the Neo4J language and are embedded in the Rover 3 / ADIM framework:

Text = 'text'

Attribute of type string 'Text' equals the exact value

Text STARTS WITH 'text'

Attribute of type string 'Text' starts with the value 'text'

Text ENDS WITH 'text'

Attribute of type string 'Text' starts with 'text'

Text CONTAINS 'text'

Attribute of type string 'Text' contains the pattern 'text'

Number ==, >, < integer | real number

Attribute of type real named ‘Text’ meets the numerical criteria of either ‘==’, ‘>’, or ‘<’

7.1.3 Ontologies

The starting point for any ontology is the ontologyroot node type. Neo4J only supports two native types, relationships and nodes. Though not a Rover3 primitive, we use the ontologyroot as a grammatical starting point for the creation of any ontology.

A dynamic ontology may contain many ontology roots. The ontology root defines the abstract starting point for any model. Every entity that will be referenced in other areas of a dynamic ontology needs an ontological root somewhere in the template store. This root defines the basic structure for that entity. Any additional references to that entity always reference back to the ontological model – the model is the set of properties, relationships, and actions that are applied by default to that entity. This allows Rover3 to instantiate such a model whenever needed. It also allows any other reference to the ontological root for that entity type to be referenced later in the dynamic model. Ontology roots are also used to navigate the various models in the GUI as well. When browsing the template store only ontological roots are displayed as starting points.

The following example creates the ontological model of a person:

```
CREATE (m:PersonOntologyroot:Person:person:entity {first_name:’’,
last_name:’’, id:’’, ssn:’’})-[:worksAs {situation_contains_text: ‘professional’,
situation_exists: ‘careerQuery’}]->(o:Profession:profession:entity {title:’’})-
[r:employedAt]->(p:Company:company:entity {name:’’})
```

The earlier example also created the entity Person1 which was also a person. When instantiating the Person1 model, Rover3 looks at the label person and checks the template store to see if an ontological root exists. If the above was defined, every instantiation of a person would automatically expand all relevant relationships and linked nodes according to the rules of the ontology root.

Other instances of ontological roots can simply reference the root through the 'isAnInstantiationOf' keyword. Doing so allows one to re-use the model within the template store. Primitives that reference an ontological root via this term have all attributes and default entity-relationship pairs copied to the instantiated one in the context store.

7.2 Variable Relationships

A relationship in an ADIM model is one of two types – it either always exists (or is instantiated) when one of the two linking nodes exists in the context store or it is subject to criteria of the current context store and its existence is variable. These variable relationships are the easiest and arguably most natural components of a dynamic ontology. Vast models for a given domain might have many relationships that are only deemed relevant and instantiated as the situation demands it. The variable relationship feature allows these models to be vast yet ensures only relevant ones propagate to the context store. Within the context store, as different primitives exist, have labels of a particular type, or attributes are set, relationships are instantiated or removed as the relevance dictates.

7.2.1 Types

There are 2 drivers that dictate when a relationship should exist: the current situation(s) or attributes that are set for any given entity. These rules can be encoded in the dynamic ontology itself. When such conditions are met in the instantiated model the rules fire, resulting in the newly formed (or removed) relationship and subsequent node. Any subsequent nodes that are required due to the existence of the newly linked node also expand and are placed in the context engine. An exhaustive search (pruned at each relationship when deemed not relevant) continues to all node-relationship pairs that branch out from the new node.

A situation based variable relationship is one where the attribute of the relationship dictates what situations must be present for the variable relationship to be instantiated. Discrete situation names or names that match patterns/words can trigger the variable relationship.

The variability of the relationship existence can also be based on entity status. As an entity and the status of an entity can essentially constitute a situation it is only natural that this also be a driver for relationships. Like the relationship and situations where the discrete names or text patterns of names can trigger the creation of relationships, the same applies for entities.

7.2.2 Variable Relationships Grammar

Making a relationship relevant based on a situation existing

situation_exists: 'situation_name

Optional attribute assigned to a relationship primitive type. The specific situation must exist in the situation graph and be active for the trigger to fire. As situations can be hierarchical, any one of the situations whose label is an exact pattern match that may exist in the hierarchy.

The following example creates the relationship *containsBloodPressureRecords* only when the situation 'carAccident' exists:

CREATE (a:HealthRecords:healthrecords:entity)-
[:containsBloodPressureRecords {**situation_exists**: 'carAccident'}]-
>(b:BloodPressureRecords:bloodpressurerecords:entity)

situation_contains_text: 'user_defined_text'

Optional attribute assigned to a relationship primitive type. A situation containing the user_defined_text must exist in the context store for the relationship to be instantiated. As situations can be hierarchical, any one of the situations whose name contains the pattern match that may exist in the hierarchy.

The following example creates the relationship *containsBloodPressureRecords* when a situation that has the pattern 'accident' exists. Thus, a situation such as 'carAccident' existing would cause this variable relationship to be instantiated:

CREATE (a:HealthRecords:healthrecords:entity)-
[:containsBloodPressureRecords {**situation_contains_text**: 'accident'}]-
>(b:BloodPressureRecords:bloodpressurerecords:entity)

Making a relationship relevant based on entity composition

entity_attribute_criteria: 'attribute test'

Optional attribute applied to relationship primitive type. For any given entity if the general primitive attribute match is true the relationship will be considered relevant. See section 6.1.2 for an explanation on what a valid attribute test is.

The following example sets the *hasKnownLastLocation* relationship for the entity *car* when the attribute *stolen* is set to true:

CREATE (m:CarOntologyroot:Car:car:entity {vinNumber: '', licensePlate: '', isStolen: '', model: '', make: '', year: '', preCacheTag: 'Action: GET http://localhost:8080/Rover3/tagQueryDatabase m.licensePlate m.isStolen'})-[:hasLastKnownLocation {***entity_attribute_criteria***: 'car.stolen = yes'}]->(o:KnownLocation:entity)

The following example would instantiate the hasCriminalRecord relationship for the entity Person when the attribute BAC is > 0.08:

CREATE (p:Person)-[r:hasCriminalRecord {***entity_attribute_criteria:*** 'BAC > 0.08'}]-(r:criminalRecords)

7.3 Enriching / pre-caching information based on current information

One of the tenets of a context aware system defined earlier was to proactively fetch data and expand models. The variable relationships outlined in section 7.2 accomplish that tenet. We now look at the notion of pre-caching information: collecting relevant and real-time information for an instantiated model as defined by the rules in the dynamic information store.

There are two triggers that can initiate a pre-caching process. The first is when certain model criteria are met upon instantiation. As template stores are instantiated and initial information is provided or exists within the situation graph, an action can take place to fetch information from a data source. This is a one-time action that auto-populates the various attributes of a node or relationship. The grammar requires a source, allows for the use of HTTP protocols and methods to interact with end services, and can send data as a part of the call.

The second way Rover3 can pre-cache is through active monitoring. Attributes can be flagged for constant updates. Similar methods exist for monitoring as above, the only difference is the monitoring verb ensures information is kept up to date. This is best used for resource tracking or information that has a lot of churn. Whenever any of the source variables are updated the monitor action fires. Using the monitor verb alone does not auto-populate the attribute. Doing so requires the attribute to receive its first value through external/manual means such as the web API.

Pre-caching has a subset of grammar for each verb. These pre-caching verbs are actions that are given to the Activity Manager. The activity manager processes the single or recurring actions needed to retrieve external information from Rover3. Both preCacheTag and monitor commands can exist on a single primitive/attribute tuple. None of the actions fire if any of the required variables are null. The activity manager queues the actions either until they fire (preCachTag) or perpetually until the parent entity of the attribute to be populated is removed from the situation graph or made inactive.

7.3.1 Grammar

One-time pre-caching

preCacheTag ‘HTTP-method endpoint [variable1 variable2 ...] target-attribute’

Optional attribute for any entity or relationship. Requires a valid stanza to follow. The target attribute is set/updated with the information that is returned from the endpoint. The string of variables is used as parameters in the URI.

Continual pre-caching

Monitor ‘HTTP-method endpoint [variable1 variable2 ...]’

Optional attribute for any entity or relationship. Requires a valid stanza to follow. The target attribute is set/updated with the information that is returned from the endpoint. The string of variables can be used as parameters in the URI.

7.3.2 Formatting of Pre-Caching grammar

Valid methods Rover3 supports:

HTTP

GET

POST

Endpoint options:

Any valid HTTP URI will work as a valid endpoint.

Variables:

Mandatory set of variables to be sent to an endpoint. When the endpoint is an HTTP URI the variables represent URL parameters to be added to the end of the URL in a REST like fashion. There is no limit on the number of variables that can be declared for HTTP endpoints. The last variable is the attribute in the ontology that is set with the return value from the HTTP call. HTTP calls must return a valid type for use with Cypher attributes. When referencing existing information via the model, use Cypher notation to identify what information is valid. The standard lable:attribute tuple suffices. Use the throw-away single-letter label for this reference.

Target-attribute:

This is the primitive-attribute tuple that is to be updated through the defined action. This parameter must be in cypher notation and cannot be a static reference.

7.3.3 Example

The following example creates a simple car ontology that uses the preCacheTag verb. The resulting HTTP request would be a GET of http://localhost:8080/Rover3/tagQueryDatabase/mlicensePlate where *m.licensePlate* is the value of the license plate of the car on the other side of the instantiated node. The isStolen attribute is then updated once the action fires.

CREATE (m:CarOntologyroot:Car:car:entity {vinNumber: ‘’, licensePlate: ‘’, isStolen: ‘’, model: ‘’, make: ‘’, year: ‘’, **preCacheTag: ‘Action: GET http://localhost:8080/Rover3/tagQueryDatabase m.licensePlate m.isStolen’**})-[:hasLastKnownLocation {source_node_attribute_stolen: ‘Yes’}]->(o:KnownLocation:entity)

The following example monitors the location of a car and updates the county jurisdiction each time a new location is provided:

CREATE (m:CarOntologyroot:Car:car:entity {positionLat: ‘’, positionLon: ‘’, IsInCounty: ‘’, vinNumber: ‘’, licensePlate: ‘’, isStolen: ‘’, model: ‘’, make: ‘’, year: ‘’, **monitor: ‘Action: GET http://localhost:8080/Rover3/countyLookup m.positionLat m.positionLon m.isInCoutny’**}) – [:hasLastKnownLocation {source_node_attribute_stolen: ‘Yes’}]->(o:KnownLocation:entity)

7.4 Collection of Actions Goal-oriented entities can take

Context aware systems and dynamic ontologies were created to manage information as situations evolve. Every situation has at least one end goal, some of which can be achieved through discrete activities. When we talk about activities and goals we first must understand the domain of both as well as how Rover3 models these primitives. This is accomplished by modeling the individual activity steps (or discrete actions) as well as an end goal. We will look at each type now, how the activities are relevant for a given situation or an entity/attribute tuple, and then explore the grammar that accomplishes this.

7.4.1 The Activity

An activity is a collection of graph nodes where each node represents a state of that given activity. This chain of states is stored in the ontology. This holds true both in the template store and the activity store. The activities are atomic themselves. Activities are generated either implicitly through states of the situation graph (a situation or some combination of entities/attributes exist) or explicitly through the ontological structure of entities or situations. An implicit trigger is one that is encoded in an activity primitive within the template store. Such triggers require the activity manager to query the template store every time and update comes in as it must look for uninstantiated activites to see if the composition of the context store has a state that warrants the creation of an activity. This is a rather expensive search through the template store but allows activities to be re-used through multiple ontologies and maximizes on hierarchical situations or general entities. Explicit definition is more efficient as triggers for such activities are defined in certain entities/situations and the ability to ascertain when the activity needs to fire requires only monitoring the current instantiated entities.

To clarify the difference between explicit and implicit definitions let us consider two different ways the trigger of a single activity can be modeled. Let us supposed the action of getMedicalCare that would be initiated if any entity has the attribute ‘Injured’. This is an implicit definition for an activity and that trigger would be defined in the structure of the activity in the ontology itself. In comparison, suppose getMedicalCare were to be instantiated using the ‘newActivity’ verb when a specific person entity has the attribute ‘Injured’. In this case since the trigger for the activity is defined within an individual primitive other than itself in the template store it is considered explicit.

Any activity requires at least one entity to be linked to the activity. In order for any action to be performed onto an entity, that entity must exist in the situation tree before the activity can be instantiated. Entities are not the only thing that can be required for an activity to exist – a situation, relationship, or any other primitive can also be required for the activity to exist. Any primitives the activity mutates or enriches information for requires the activity to be linked to those primitives. All required primitives are linked via a 'requires-these-primitives' relationship. All explicit and implicit primitives that fire the creation of the activity must exist in the situation graph. Some primitives can be required but not exist at the time of the activity being fired. Those primitives must be created at the time of the activity generation and will be linked to the activity. For example, the activity getMedicalCare might require someone to provide on-site medical response.

Linking occurs by copying the nodeID of the instantiated neo4j node into the attribute of the node that represents the required primitive. The Rover3 framework does this automatically as a part of interpreting and actioning the grammar. Activities are waterfall in nature, so there is ever only one state that is current at one time. A master node must exist in the template store. This node is the reference to the activity and links to all activity states through the ***isAnActivityStateOf*** relationship. This relationship originates at the individual state and terminates at the master node. This master node and all states are copied to the context store upon instantiation and acts as the pointer to the current state of the activity.

Each activity has an order. The first activity state will have the attribute ***firstActivityState*** set to true. The ***isSucceededBy*** relationship must follow from that state, through all of the states until it reaches the goal.

Every activity state has a few default characteristics that must exist in the various nodes or relationships for Rover3 to process. If these attributes or characteristics are missing in the template store Rover3 will not process the activity. Each activity state has a Boolean attribute ‘met’ that conveys if that state has been reached in the activity stream. Additional mandatory grammatical items are outlined below in the grammar index.

7.4.2 Goals

Goals are a primitive that can optionally be the final node in an activity chain or a generic goal for a situation. Goals represent a desired end state. The context stores tracks all goals for a given situation. Achieving all goals warrants the situation to no longer be active in the context store. When the goal is a terminal node in an activity chain, the act of reaching such a state prior to visiting previous states terminates the activity as the desired effect of the activity has been achieved. Every goal primitive also has the mandatory Boolean attribute ‘met’ and a label which is the name of the goal. Additional mandatory attributes include ‘time_goal_created’ and ‘time_goal_met’.

Activity states are reached when defined criteria are met. All state transition rules are defined within the activity itself and must be implicit. The activity manager monitors the situation graph to see if the explicit criteria has been met. Once all states have been reached the goal of the activity is complete. A complete activity is kept in the activity store until the situation or entity that triggered the activity is no longer in the context store. This node is the reference to the activity and links to all activity states through the relationship isAGoalOf relationship.

Goals that are not a part of an activity are simply monitored by the context store. Such goals are defined as a part of the situation themselves. When creating situations within the template store, one must ensure each situation has at least one goal to close, either through a related activity or a stand-alone goal. All goals must either be an ontological root of the goal or a reference to the root.

7.4.3 Grammar

activity

Mandatory label for every activity master node. This informs Rover3 that the primitive is a master activity node. This can be considered the same as an ontologyroot label for entities.

activityState

Mandatory label for every activity state. This informs Rover3 that the primitive is an activity state.

Goal

Mandatory label for a goal primitive. Ontology roots must also have the *OntologyRoot suffix in the label (see below example).

hasAGoalOf

Optional situation attribute that tracks a goal for a given situation. The named goal must have an ontological root defined in the ontological store. The following example details how one can link a goal to a situation (and create a stand-alone goal):

```
CREATE (g:CarAccidentOntologyroot:CarAccident:accident:situation
{location:'', hasAGoalOf:'carAccidentClear'})
CREATE (g:carAccidentClearOntologyRoot:carAccidentClear:goal)
```

ifEntityExistsAs 'attribute test'

Optional attribute for activity master node. This attribute implicitly defines the criteria for instantiating the activity. The named entity must exist in the context store with the attribute set to the user-defined-value for the instantiation of the activity to fire. The template manager will check all activities for this rule each time the Rover3 system sends out a heartbeat that the context of the system was updated.

The following example details how the activity getMedicalHelp is implicitly triggered when the Person entity has a status that contains the pattern 'injur', allowing the status to be 'injured' or 'injury'. See section 6.1.2 for an explanation on what a valid attribute test is.

```
CREATE (a:getMedicalHelpActivity:activity {ifEntityExistsAs: 'status
CONTAINS injur'}), (b:callAmbulance:activityState {met: 'false'}),
(c:ambulanceArrives:ActivityState {met: 'false'}), (d:loadInjured:goal {met:
'false', time_goal_created: '', time_goal_met: ''})
```

IfSituationExists 'situation_name'

Optional attribute for activity master node. This attribute implicitly defines the specific situation that must exist in the context store for the activity to be instantiated. When named situation exists in the context store the trigger is engaged and the activity will instantiate. The template manager will check all activities for this rule each time the Rover3 system sends out a heartbeat that the context of the system was updated.

The following example creates the 3 steps for an activity called getMedicalHelpActivity and the activity is instantiated when the situation injuredPerson exists in the context store:

CREATE (a:getMedicalHelpActivity:activity {***ifSituationExists***: 'injuredPerson'}), (b:callAmbulance:activityState {met: 'false'}), (c:ambulanceArrives:ActivityState {met: 'false'}), (d:loadInjured:goal {met: 'false', time_goal_created: '', time_goal_met: ''})

linkNodeTo 'newNode realtionshipName existingPrimitive direction'

Optional auto-instantiation for an activity-produced entity. Entities that are created either through the instantiation of an activity or the entering of a specific state can be auto-related to existing primitives in the context store. This verb links the newly created node to a required node for the activity with the named relationship. The existing primitive must already exist in the context store for this to fire. The direction variable is one of either *fromNew* or *toNew*. The optional keywords ***triggeredSituation***, ***triggeredRelationship***, and ***triggeredEntity*** allow an ontology author to link a new node to a node that triggered the activity.

newActivity ‘Activity_Name’ [attribute test]

Optional explicit trigger to create activities for entity, relationship, or situation. The user-defined activity name will be instantiated when the respective primitive is instantiated. The named activity must exist in the template store or it will throw a run-time error.

The following example is an explicit definition of an activity that will be instantiated through the definition of a Person of a particular type. This requires that the getMedicalHelpActivity exist in the template store or the verb will not trigger any activity in Rover3:

```
CREATE (n:CarAccidentOntologyroot:ontologyroot {newActivity:
‘getMedicalHelpActivity’})
```

newEntity ‘Entity-name’

Optional attribute for an activity state to create a new entity when the state is active. This verb will create an entity of specific name provide it exists in the template store.

requiredEntity ‘true|false’

Mandatory Boolean attribute for each primitive that is required for the activity. If this is true that entity must exist at the time of instantiation. If this is false the entity will be created at the time of instantiation.

updateGoalWhen ‘primitiveLabel.attibute: user-defined-value’

Optional attribute for the last activity state. This primitive / attribute tuple must be met in the context store for the goal to be met. When this happens the goal's attribute *met,* is set to true. The primitiveLabel:attribute variable must be a in valid Cypher node/ attribute syntax. The optional keywords ***triggeredSituation***, ***triggeredRelationship***, and ***triggeredEntity*** allow an ontology author to link a new node to a node that triggered the activity.

updateStateWhen 'primitiveLabel.attibute: user-defined-value'

Mandatory attribute for every activity state. This primitive / attribute tuple must be met in the context store for the individual state to be considered met. The primitiveLabel:attribute variable must be a in valid Cypher node/attribute syntax. The optional keywords ***triggeredSituation***, ***triggeredRelationship***, and ***triggeredEntity*** allow an ontology author to link a new node to a node that triggered the activity.

The following example shows how a simple three-state activity of getMedicalHelp is defined in the template store. Note that this example is implicit as the activity fires when a situation exists of a certain type. The new Person entity FirstResponder is created and is a re-use of the Person ontology. This FirstResponder is connected to the situation itself and is created in the context store at the time the activity is instantiated:

> CREATE (a:getMedicalHelpActivity:activity:ontologyRoot {***ifSituationExists***: 'injuredPerson', ***ifEntityExistsAs***: 'status CONTAINS injur'}), (b:callAmbulance:activityState {***met***: 'false', **firstActivityState**: 'true'}), (c:ambulanceArrives:ActivityState {***met***: 'false'}, ***updateStateWhen*** 'Ambulance.status called'), (c:ambulanceArrives:ActivityState {***met***: 'false', ***updateStateWhen***: 'Ambulance.location 'onScene'}), (d:loadInjured:goal {***met***:

‘false’, ***time_goal_created***: ‘’, ***time_goal_met***: ‘’, ***updateGoalWhen*** ‘triggeredEntity.location inAmbulance’}), (e:FirstResponder:person:entity {***requiredEntity***: ‘false’, ***linkNodeTo***: ‘e providesMedicalSupportTo triggeredSituation fromNew’, ***linktNodeTo***: ‘e assists triggeredEntity’})

MATCH (f:getMedicalHelpActivity:activity {***ifSituationExists***: ‘injuredPerson’, ***ifEntityExistsAs***: ‘status CONTAINS injur’}), (e:FirstResponder:person:entity {***requiredEntity***: ‘false’, ***linkNodeTo***: ‘e providesMedicalSupportTo triggeredSituation fromNew’, ***linktNodeTo***: ‘e assists triggeredEntity’})
MERGE (e)-[:isAResponderWith]->(f)

MATCH (m:PersonOntologyroot:Person:person:entity {first_name:'', last_name:'', id:'', ssn:''}), (e:FirstResponder:person:entity {***requiredEntity***: ‘false’, ***linkNodeTo***: ‘e providesMedicalSupportTo triggeredSituation fromNew’, ***linktNodeTo***: ‘e assists triggeredEntity’})
MERGE (e)-[:isAnInstantiationOf]->(m)

MATCH (f:getMedicalHelpActivity:activity {***ifSituationExists***: ‘injuredPerson’, ***ifEntityExistsAs***: ‘status CONTAINS injur’}), (g:callAmbulance:activityState {***met***: ‘false’})
MERGE (f)-[:***isAnActivityStateOf***]->(g)

MATCH (f:getMedicalHelpActivity:activity {***ifSituationExists***: ‘injuredPerson’, ***ifEntityExistsAs***: ‘status CONTAINS injur’}), (h:ambulanceArrives:ActivityState {***met***: ‘false’})
MERGE (f)-[:***isAnActivityStateOf***]->(h)

MATCH (f:getMedicalHelpActivity:activity {***ifSituationExists***: ‘injuredPerson’, ***ifEntityExistsAs***: ‘status CONTAINS injur’}), (i:loadInjured:goal {***met***: ‘false’, ***time_goal_created***: ‘’, ***time_goal_met***: ‘’})
MERGE (f)-[:***isAnActivityStateOf***]->(i)

MATCH (f:callAmbulance:activityState {***met***: 'false'}), (g:ambulanceArrives:ActivityState {***met***: 'false'})
MERGE (f)-[:***isSucceededed By***]->(g)

MATCH (f:ambulanceArrives:ActivityState {***met***: 'false'}), (g:loadInjured:goal {***met***: 'false', ***time_goal_created***: '', ***time_goal_met***: ''})
MERGE (f)-[:***isSucceededBy***]->(g)

7.5 Dynamic Situations

Situations alone do not dictate what is relevant. The information in any setting can change, resulting in an evolving situation, a situation to no longer be present, or become an entirely new situation all together. Rover3 has two ways of modifying the situation present in the context store. As ADIM grammar provides the ability to evolve relevant information at runtime, we allow the information in the context store to inform Rover3 about how to modify the given situations. Additionally, the Rover3 REST API allows for users or external systems to modify the situation (to be discussed in section 6.5 – API reference manual). The setSituation ADIM verb can be applied to any primitive. Without additional constraints, the existence of that verb instantiates the situation when the entity itself is instantiated in the context store. Instantiating a situation allows the situation manager to merge, refine, or broaden the current situation in the context store as a result, but that is left up to the processing in the situation manager. An ontology author has the option to dictate additional constraints on the attribute of the entity as well.

7.5.1 Grammar

setSituation ‘Situation_name [attribute test]’

Optional attribute for any entity in the template store. The situation name must be a valid situation in the template store for the trigger to fire. Optional constraints allow for the evaluation of an entity’s attribute. The situation manager will track the rule once the owning primitive is instantiated. The rule will always be valid and possible rule as situations can churn. If there is no attribute test, the act of instantiating the primitive into the context or activity store will set the situation. See section 6.1.2 for an explanation on what a valid attribute test is.

Note that there can be more than one situation active at any given time. The situation manager will handle the merging and aggregation of situations. The ADIM rules simply determine what relevant situations evolve over time and trigger Rover3 to instantiate them.

The following example sets the medical emergency situation when the Person entity has a status representing an ‘injured’ state. Note this setting is such that the instantiation of the Person entity does not set this situation but rather the status is monitored and only when the status reflects an inury does that situation become present:

CREATE (n:PersonOntologyroot:ontologyroot {***setSituation***:
medicalEmergencySituaiton ‘status CONTAINS injur’})

7.6 Web API

Creating ontologies is what allows Rover3 to work with models. Placing them in the template store is not enough to instantiate a model. The Web interface is what allows external systems or users to actually instantiate situations, manually extend or shrink situation graphs (by adding or removing primitives), search the entire context store, enrich the information that exists in the context store to date, or even remove and delete any and all primitives that exist. The API is a RESTful interface and calls are easily crafted through simple rules to modify or view any information in the context or activity store. It is the Web API that allows the second class of users defined in section 6.1 to use the Rover3 system.

Additional read-only calls exist within Neo4J that allow UI components to browse graphs. We leverage these calls and modified the popoto package to provide users browsing mechanisms to view the ontologies and situation graphs that live in the template store and context store respectively.

7.6.1 Retrieving context

As Rover3 is a context-aware system, we chose to refer to information and relevance as context when naming the API calls. The first set of API calls relates to obtaining context – or information that exists in the context store. While items also exist in the activity store, what does exist in the activity store are the rules and triggers for the activity manager. Activities and states are returned by browsing the context store. All 'get' functions essentially query the context store.

7.6.1.1 Commands for retrieving context

The *getcontext* command retrieves specified aspects of information for a primitive. You can also list the labels or IDs of all primitives that are associated with a situation. All data is returned via a JSON object. The *getcontext* command is used both to discover what exists in the context store as well as to retrieve discrete values/components of the structure.

getContext/Situations/[lables | ids]

The GET call returns all situations that exist in the situation graph. The user must specific labels or ids to be returned.

Example:

http://localhost:8080/Rover3/getContext/Situations/ids

getContext/Situation/id/[entities | relationships | activities]

The GET call returns all entities and relationships that exist in the situation graph. Optional delimiter of *entities*, *relationships*, or *activities* will list only the respective primitive type.

getContext/Situation/ID/primitive/[lables | ids | all]

The GET all returns the items that exist in the situation graph of type *primitive*. Valid options are [labels | ids]. The use of this call is to be able to iterate on the specific IDs for future calls or identifying labels within the situation graph. The following primitive types are supported:

Entity

Relationship

Activity

The following example returns all ID's for a situation:

http://localhost:8080/Rover3/getContext/Situation/65/entity/ids

Returns:

```
[{"(id(m))":65},{"(id(m))":66},{"(id(m))":67},{"(id(m))":68},{"(id(m))":
74},{"(id(m))":75},{"(id(m))":69},{"(id(m))":71},{"(id(m))":72}]
```

getContext/PrimitiveType/ID/values

The GET call returns the items and values as described. The following primitive types are supported:

Entity

Situation

Relationship

The following *values* are supported and one of each must be a part of the call:

Properties

Relationships (only for situation and entity)

The following example returns all of the attributes of the entity with an ID of 66:

http://localhost:8080/Rover3/getContext/Entity/66/properties

Returns:

{“properties(n)”:{“last_name”:””,”id”:””,”model_id”:”66”,”first_name”:”
”,”ssn”:””}}

7.6.2 Creating context

The *createContext* API call allows users to create primitives in the context store. All default relationships, related entities, and activities will be created when creating situaitons. Creating a situation will create a new situation graph and all default (or those considered relevant) relationships, entities, and activities for the situation in the ontological store will auto-instantiate.

When creating primitives (other than situations) one must later reference the situation graph the primitive is to be a member of. This can be done directly by linking the entity to the situation itself or another entity that is already in the situation graph. The *updateContext* command in the next section provides the tooling needed to associate the newly created primitive with the current situation and completing the relevance. We provide users the ability to create entities and relationships based on the ***system helps the user*** tenet – without this function, users would only ever consume information Rover3 considered relevant. As Rover3 is meant to be an aid and not an authoritative source of relevant context, we must provide this mechanism for users to manually evolve the context store.

There is no need to link situations to existing situations as the situation manager will merge/refine situations as warranted. The *updateContext* call in the next section does allow Rover3 users to manually manage situations as warranted. The createContext call does, however, have the ability through optional commands to allow one to create an entity and relate it to a situation or entity that already exists in the context store.

There are no WEB API commands for creating activities. In order for activities to exist they must be instantiated from the template store when the situation graph warrants the activity to be relevant.

7.6.2.1 Commands for creating context

createContext/Situation/situation_name

This POST call creates a situation with the named **situation_name** as provided by the user. The situation name does not have to exist in the template store, but without such an entry in the template store the situation is just a primitive with the label provided by the user. Entities, relationships, and other primitives can further be added to the situation graph, but any situational rules will not apply until the situation is one that represents something in the template store.

The following example creates a situation called carAccident:

http://localhost:8080/Rover3/createContext/Event/CarAccident

The output of the call is the following text:

```
Created new situation with System id of 77
Finished creating Context...
```

createContext/Entity/entity_name

This POST command creates the names entity as provided by the user defined **entity_name** parameter. If the entity type exists in the template store all attributes of the entity will be created. The entity does not have to exist in the template store. When a user creates an entity that does not exist in the template store it will simply be a node with a label of **entity_name** and not inherent any attribute or relevance rules (that linking to a situation or entity via *updateContext* would later accomplish).

The following example creates the entity 'Person':

http://localhost:8080/Rover3/createContext/Entity/Person

The output of the call is the following text:

```
Created new entity with System id of 78
Finished creating Context...
```

createContext/Entity/entity_name/LinksTo/primitive_ID/Via/relationship_name

This POST command creates the named entity **entity_name** and links it to the primitive ID **primitive_ID** by creating the relationship **relationship_name**. The entity and relationship need not exist in the template store, though if the entity and relationship do exist in an ontology in the template store the context engine will expand any variable realtionships and instantiate any activities or other primitives as deemed relevant.

The following example creates an entity called ‘Person’ and links it via the relationship ‘providesMedicalAssistanceTo’ to an existing entity with an ID of 78:

http://localhost:8080/Rover3/createContext/Entity/Person/LinksTo/78/Via/providesMedicalAssistanceTo

The output of the call is the following text:

```
 Created new entity with System id of 79
 Created new relationship with System id of 80
 Linked entity 79 -> entity 78 via relationship id 80
Finished creating Context...
```

7.6.3 Updating context

Updating context is likely the most used of the web API commands. It is through this command that information specific to the situation and collection of entities is actually entered. The *createContext* command creates and links primitives within the context storein an effort to deliver a relevant graph for the situation at hand. The updateContext family of commands is how actual information that is observed in the real world (the attributes) is entered for the situation. For example, after using *createContext* to create a car accident, updateContext can set the location for the accident, the name of one of the people in the accident, or update the health status or medical records for one of the injured.

The updateContext command family should be used as situations unfold and continue to evolve and is the mechanism to keep relevant information in the situation graph.

As information is entered, the Rover3 system constantly monitors the part of the context store and activity store for that situation graph. As information is entered or updated into the situation graph via updateContext calls, Rover3 will constantly check to see if the situation graph needs to expand or contract. The ADIM language described is what facilitates this. Through proper ontology creation and use of the Web API calls, automatic relevance recognition is accomplished through Rover3.

Unlike createContext there are multiple updateContext commands. As primitives can have the same label we sometime need more specific ways to update one. *updateContextById* allows a user to update a primitive by ID and is specific. A more general approach is had as *updateContextByLabel* allows one to reference the primitive by a set of labels.

7.6.3.1 Commands for updating context

updateContextById/nodeID/attribute/value

This PUT command updates a primitive that is identified by the nodeID. The attribute and value are user defined and can be any valid text. Attributes do not need to exist as default attributes in the template store and thus do not necessarily need to be defined in the ontology itself.

The following example sets the ‘ssn’ attribute to ‘123456789’ for the primitive whose ID is 17:

http://localhost:8080/Rover3/updateContextByID/17/ssn/123456789

The output of the call is the following text:

```
The ssn attribute was updated to 123456789 for primitive ID 17
```

updateContextByLabel/primitive/label_list/attribute/value

This PUT command updates a primitive that is identified by a list of labels. All labels must exist for the primitive. The list of labels must be unique, that is, there must be no other primitive with that label combination in the information store. The label list is unbounded in length and delimited by forward slashes as the URI is extended. The attribute and value are user defined and can be any valid text. Attributes do not need to exist as default attributes in the template store and thus do not necessarily need to be defined in the ontology itself.

The following example sets the 'ssn' attribute to '123456789' for the primitive whose label set is Person:Nick:

http://localhost:8080/Rover3/updateContextByLabel/Person/Nick/ssn/123456789

The output of the call is the following text:

```
The ssn attribute was updated to 123456789 for
primitive whose label set was Person:Nick
```

The following examples sets the location_long and location_lat attributes for the primitive Situation:CarAccident:TwoCarAccident

http://localhost:8080/Rover3/updateContextByLabel/Situation/CarAccident/TwoCarAccident/location_lon/38.78827634

The output of the call is the following text:

```
The location_lon attribute was updated to 38.78827634 for primitive
whose label set is Situation/CarAccident/TwoCarAccident
```

http://localhost:8080/Rover3/updateContextByLabel/Situation/CarAccident/TwoCarAccident/location_lat/-140.345234

The output of the call is the following text:

```
The location_lat attribute was updated to -140.345234 for primitive
whose label set is Situation/CarAccident/TwoCarAccident
```

7.7 Browsing Context and Ontologies

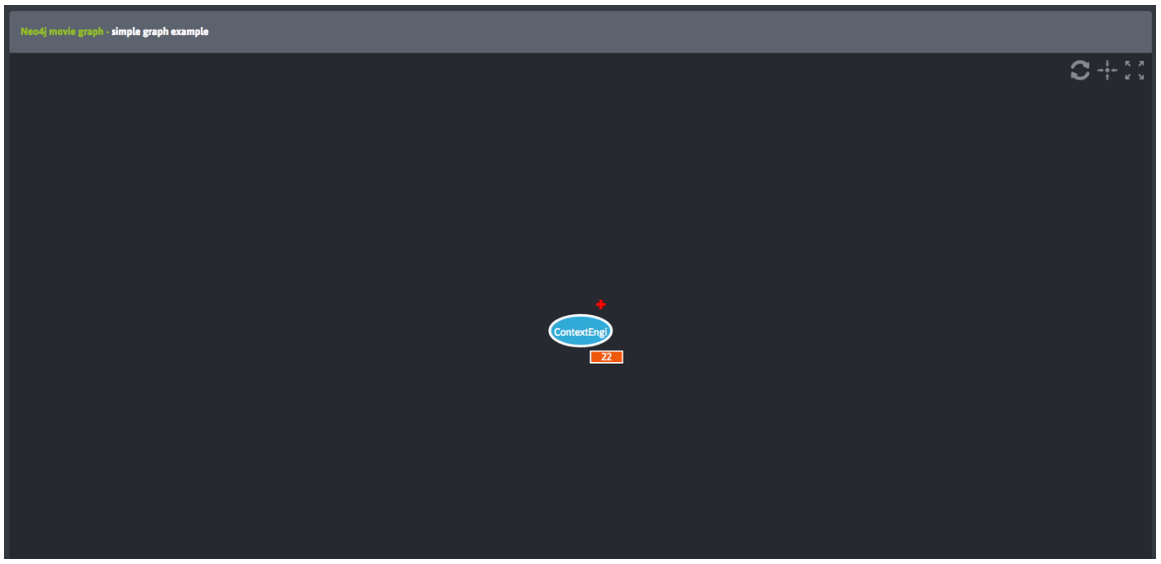
Extending popoto.js [55] we provide a front-end GUI that allows users to freely browse what ontologies exist in the template store as well as what the current situations exist in the context store. This visual guide is used to both witness how the situation graph expands automatically as Rover3 defines what parts of the ontology in the template store is considered relevant for the given situation. It also provides a mechanism to browse the ontologies in the template store instead of simply reading over the lengthy list of Cypher commands that otherwise make up an ontology.

7.7.1 Browsing the Situation Graph and the Context Store

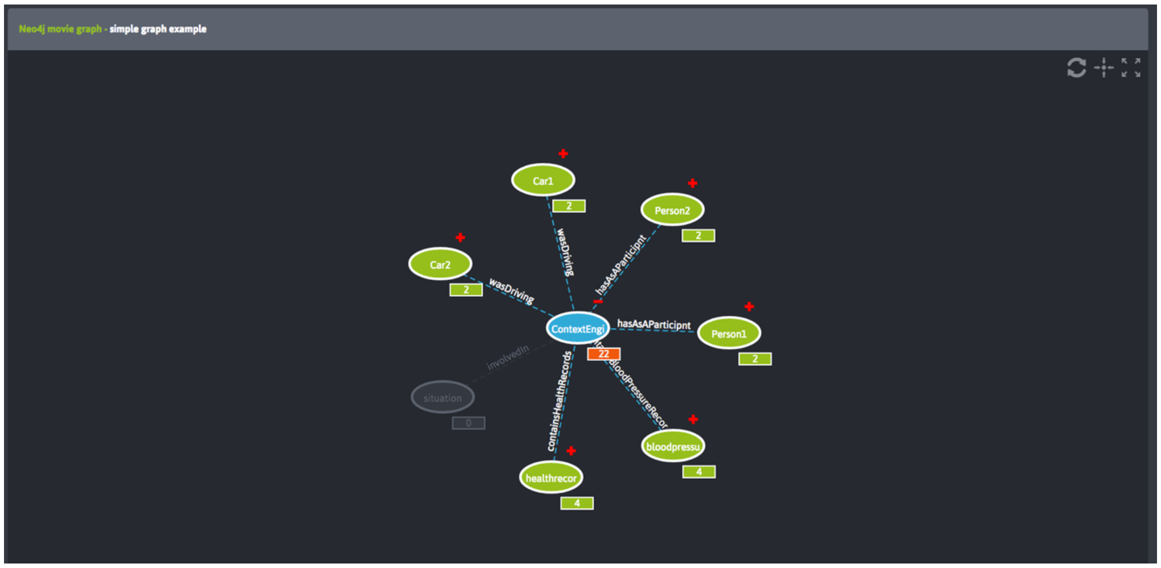
The following URL pops up a popoto :

http://localhost:8080/Rover3/currentContext.html

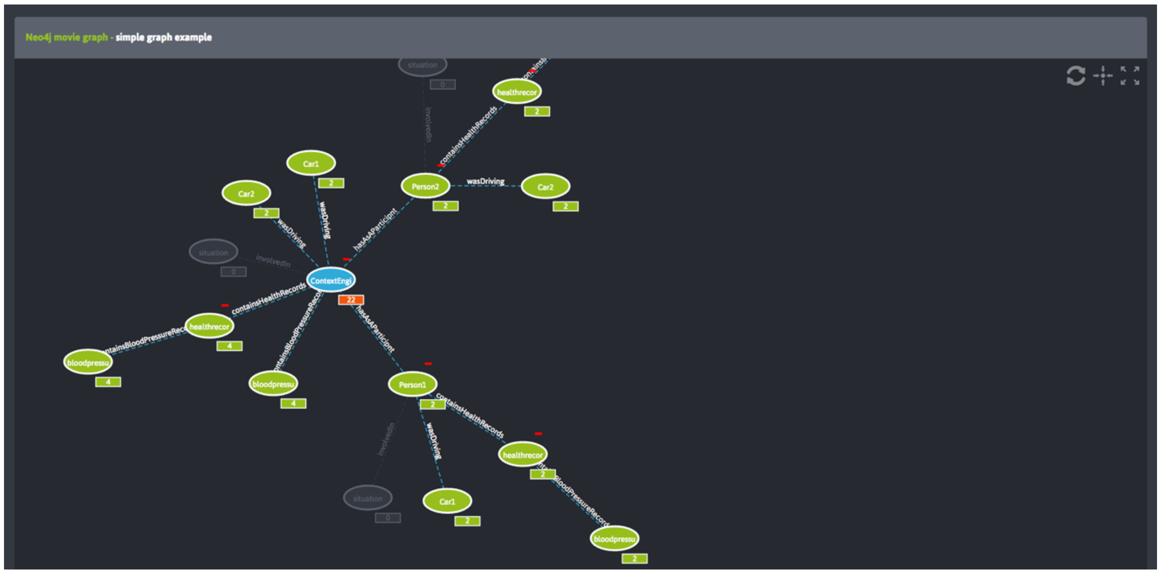
The resulting page resembles this:



We can use the built-in controls of popoto to expand the situation (referenced by ID 22) to see the following:

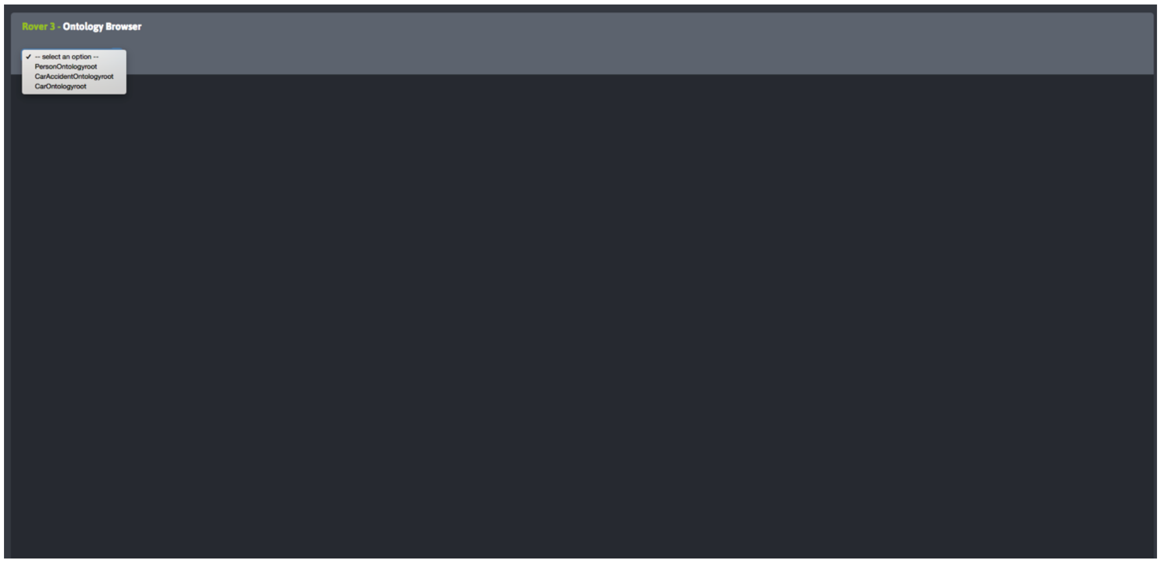


Expanding all nodes, we see what Rover3 considers to be relevant for this situation given the current information and situation that is present in the context store:

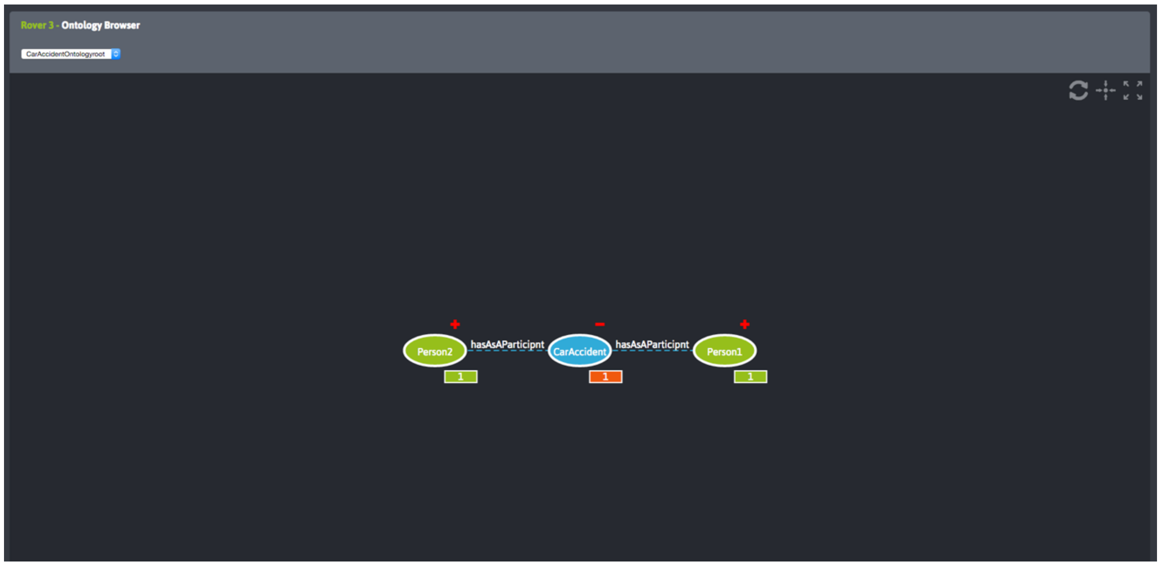


7.7.2 Browsing Ontologies and the Template Store

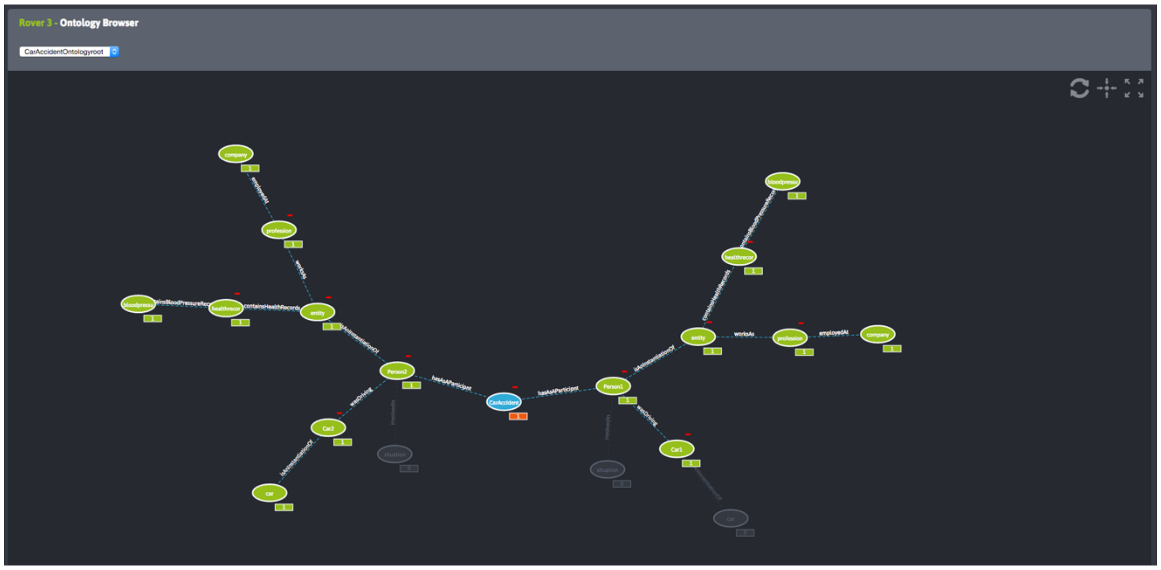
The ontology browser has similar controls as the context browser. By default, the page is empty and a pull-down allows the user to select which ontology they want to explore:



After selecting an ontology, you will see the blue node that is the seed of the ontology. In this view, we have expanded the model out one layer:



If we expand all the way out we can see all possible relationships and entities that might exist when starting from one situation:



Note that this entire graph is not what would be instantiated in the context store. Rather Rover3 would provide a relevant graph by following the rules of the ADIM grammar that exists in the ontology itself.

8 Modeling and using Rover3 to handle a car accident – A Case Study

In this chapter, we walk through an end-to-end use case where we use Rover3 and ADIM to assist with a car accident. We start by creating and re-using existing ontologies. Using ADIM we will create some of the potential models that could arise during a car accident. Using the ADIM grammar we will encode relevance in the model to ensure the only thing in the context store is a model framework that would be considered relevant. We will explore existing ontologies in the template store in an effort to minimize redundant efforts and reuse existing models. We then will craft the necessary cypher commands needed to complete the modeling for a car accident, expanding any existing models and ensuring relationships and entities that might come into play are created.

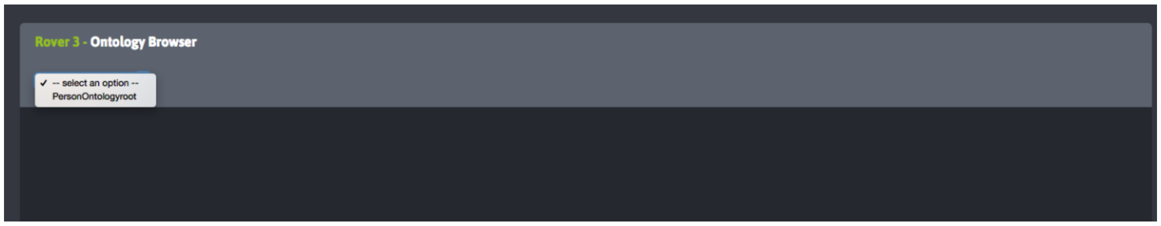
Once we feel the template store is sufficient in its ability to emulate the situation, we will use the Web API to instantiate the situation. At this point the context engine will receive a situation graph from the template manager and build a skeleton graph for the situation. There will be no real information other than the fact that a car accident has occurred. A goal will be created and tracked via the activity manager. The situation graph will be a model that is relevant to the situation and a sub graph of the larger ontology we have in the template store. At this point we will be able to view the situation in the web browser as well as the Neo4J graph browser as it has additional abilities to disclose the ADIM rules in the views.

As we use the Web API to add information to the context store, node and relationship attributes in the situation graph will be updated. As these attributes match the rules and triggers defined in ontology, the information that is considered relevant will change if warranted. This will be observed by the situation graph expanding according to the ADIM rules that exist in the ontology that has been instantiated. Activities will also be created and progress as needed, with possible goals being created or met.

8.1 Building the Ontology

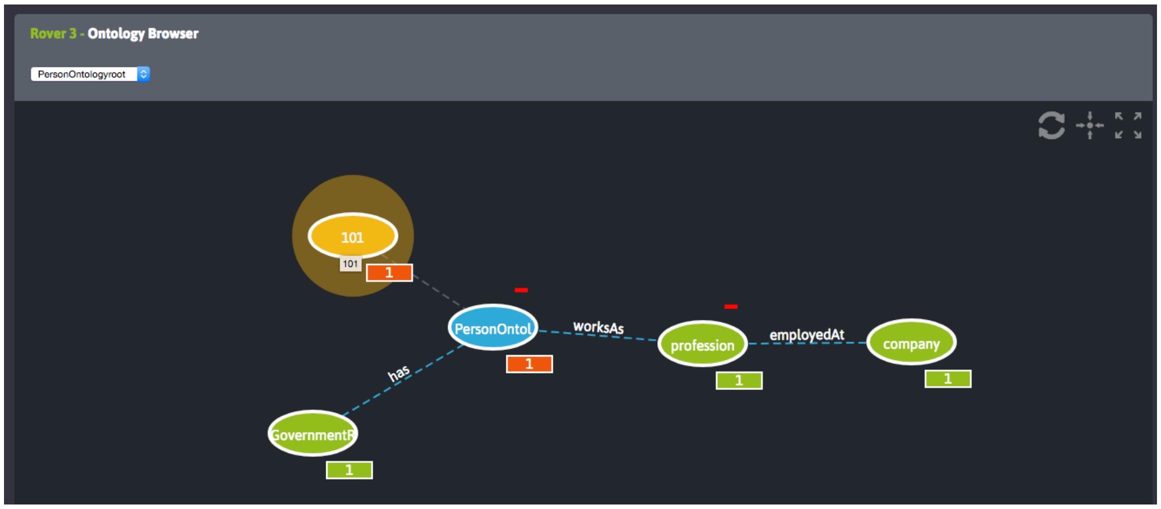
Before we start building the ontology itself we will want to see what models already exist that we could use. We have 2 different ways to browse the ontologies that are stored in the Template Store: we can use the Front-End Gui as well as the Neo4j Browser [56]. The Front-End is best for general browsing and exploration. Once the specific ID's are known, the Neo4J can be used to browse and expand the ontology as needed. An area of continued research and system design would be to expand the Front-End GUI to allow for complete ontology exploration, expansion and modification.

Exploring the GUI in our sample template store, we see an existing ontology for a person:



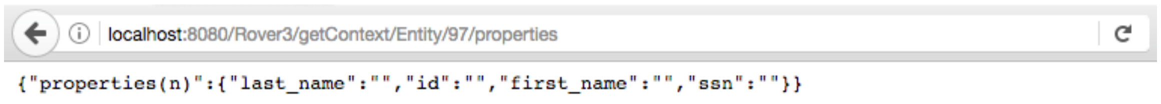
8.1.1 1 – Ontology Browser shows we have one ontology in the template store; that of the person.

Selecting the person ontology and fully expanding it we see it contains a few relationships to other items but it is rather limited for our situation of a car accident:



8.1.2 1 – Selecting the PersonOntology model and expanding the model we see there are three relationships: the person can have government records, he can work as a profession, and that profession can be employed at a company. We also see the nodeID is 101.

Here we see a person is modeled to workAs a profession and be employedAt a company. We also see this generic person can have accompanying government records. Using the individual nodeIDs we can use the getContext web API call to see what the exact properties of each node are. For example, to see what attributes are listed for the person entity in the context store we can make the following call:



```
localhost:8080/Rover3/getContext/Entity/97/properties
{"properties(n)":{"last_name":"","id":"","first_name":"","ssn":""}}
```

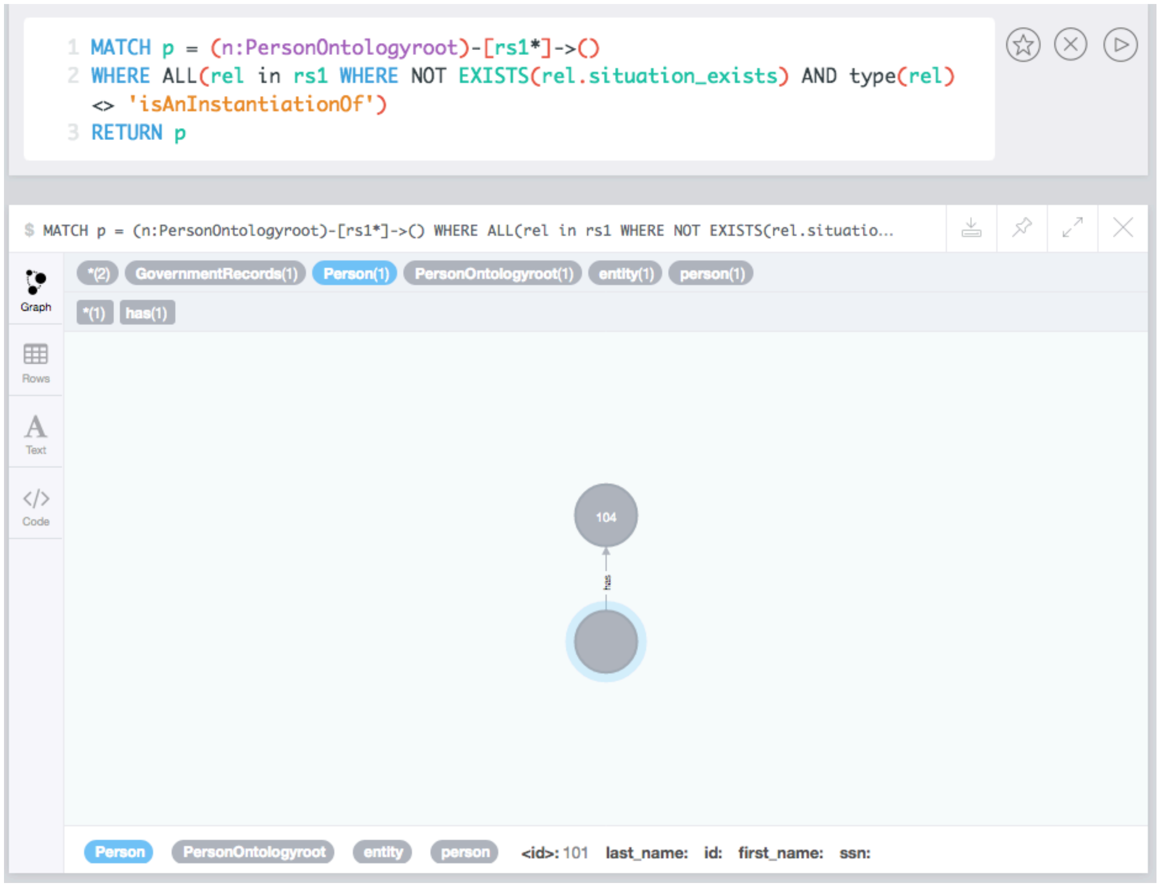
8.1.3 1 – Results of the getContext call to the id in the template store.

We can also query the context store directly via the Neo4J web browser and see what properties exist with each node and relationship. Using the following query in the browser:

MATCH p = (n:PersonOntologyroot)-[rs1*]->()

```
WHERE ALL(rel in rs1 WHERE NOT EXISTS(rel.situation_exists) AND
type(rel) <> 'isAnInstantiationOf')
RETURN p
```

We can explore the situation graph in the context store that contains the ontology for that entity. Using the isAnInstantiationOf key word we ensure we do not include references to the entity in other ontologies by omitting those relationships. Ensuring the situation_exists attribute does not exist in any relationships also filters existing dynamic relationships that exist in the ontology. Our resulting query shows us what is instantiated by default for any person. Clicking on each primitive we can see what attributes exist. Note that when we omit the filter for situation_exists we have an additional relationship present between the person entity and the profession entity. This relationship is a conditional relationship and will not automatically be instantiated when the person entity is moved to the context store. The resulting images showing the browsing of the attributes and graphs follow.

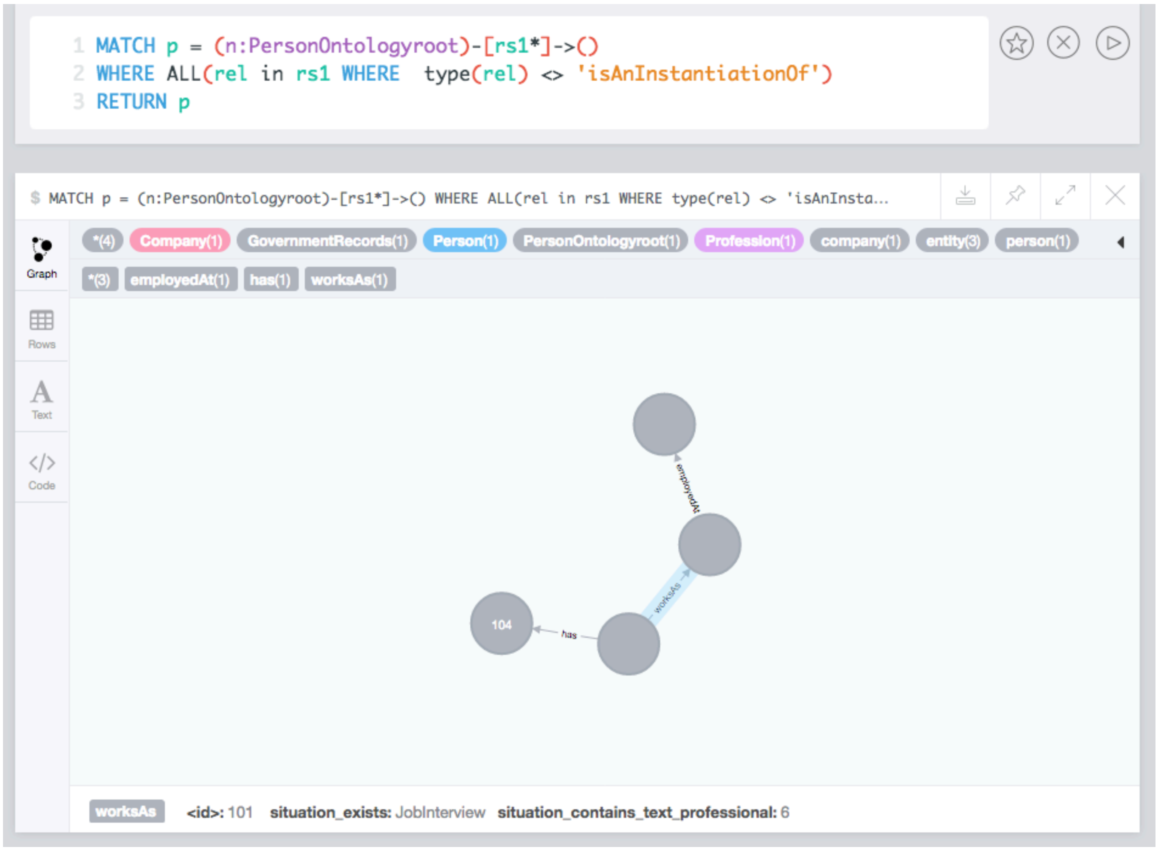


```
MATCH p = (n:PersonOntologyroot)-[rs1*]->()
WHERE ALL(rel in rs1 WHERE NOT EXISTS(rel.situation_exists) AND type(rel)
<> 'isAnInstantiationOf')
RETURN p
```

```
$ MATCH p = (n:PersonOntologyroot)-[rs1*]->() WHERE ALL(rel in rs1 WHERE NOT EXISTS(rel.situatio...
```

Person PersonOntologyroot entity person <id>: 101 last_name: id: first_name: ssn:

8.1.4 1 – Using the Neo4J browser and pruning situation_exists and isInstantiationOf relationships, we see what the default instaitaion of the person entity entails.

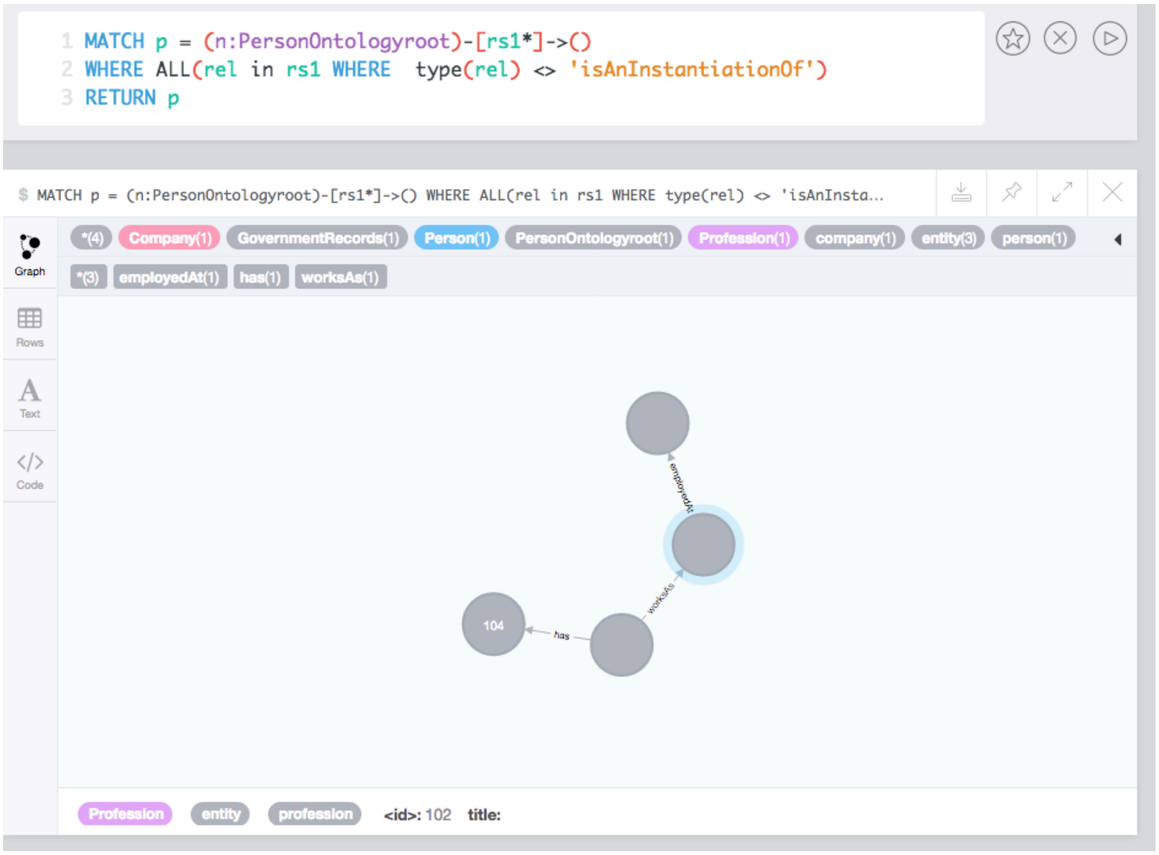


```
MATCH p = (n:PersonOntologyroot)-[rs1*]->()
WHERE ALL(rel in rs1 WHERE  type(rel) <> 'isAnInstantiationOf')
RETURN p
```

```
$ MATCH p = (n:PersonOntologyroot)-[rs1*]->() WHERE ALL(rel in rs1 WHERE type(rel) <> 'isAnInsta...
```

worksAs <id>: 101 situation_exists: JobInterview situation_contains_text_professional: 6

8.1.5 1 – Using the Neo4J browser and only pruning isAnInstationof relationships we see what the entire person ontology entails. Notice the worksAs relationship is a conditional relationship and will only be instantited if the JobInterview situation exi



```
MATCH p = (n:PersonOntologyroot)-[rs1*]->()
WHERE ALL(rel in rs1 WHERE  type(rel) <> 'isAnInstantiationOf')
RETURN p
```

```
$ MATCH p = (n:PersonOntologyroot)-[rs1*]->() WHERE ALL(rel in rs1 WHERE type(rel) <> 'isAnInsta...
```

8.1.6 1 – Using the Neo4J browser and viewing the entire person ontology we can explore the attributes of nodes that only are instaitaed if the conditional relationships are created.

As we are creating an ontology for a car accident we will want to add to the template store the necessary entities and relationships required to model the situation. We will keep this simple for the sake of the example as we model the following:

- Create the car accident ontology itself (the subgraph in the context store that must be instantiated when a car accident is instantiated).
- Define the goal and actions to take to clear the accident.
- Extend the existing person ontology in a way that can be used for the car accidents or any other situation that might involve a person getting hurt.

- Create a vehicle model (as we do not currently have one) to include actions to perform if the vehicle is stolen as well as relationships to show ownership and who is driving.
- Add all necessary contributors to the situation, referencing the person ontology in order to re-use the existing model.

Let’s start with creating the car accident itself. We will create the basic accident ontology root, a model for a two-car accident and an activity to track the goal of clearing the accident. The accident ontology root and the two-car accident situation is pretty basic:

```
CREATE (g:CarAccidentOntologyroot:CarAccident:accident:situation
{location:’’, newActivity:’clearCarAccident’})
```

And

```
CREATE (c:TwoCarAccident:CarAccident:Accident:situation)

MATCH (t:TwoCarAccident),
(c:CarAccidentOntologyroot:CarAccident:accident:situation)
MERGE (t)-[:isAnInstanitationOf]->(c)
```

Notice that the CarAccident situation automatically instantiates a clearCarAccident activity, so we should model that in the template store as well. That instantiation is not called out in the two-car accident as it is inherited by it being an instantiation of the general car accident ontology. The activity will be 2 simple steps and a goal. The steps will be annotating the accident and then clearing the accident. The goal of the scene being clear is what we will be after. The Cypher commands required to do this are:

```
CREATE (a:clearCarAccidentOntologyroot:clearCarAccident:activity),
(b:annotateScene:activityState {met: ‘false’, firstActivityState: ‘true’}),
(c:clearScene:activityState {met: ‘false’}), (d:sceneClear:goal {met: ‘false’,
time_goal_created: ‘’, time_goal_met: ‘’, updateGoalWhen:
‘triggeredSituation.itemsOnScene 0’})

MATCH (f:clearCarAccidentOntologyroot:clearCarAccident:activity),
(g:annotateScene:activityState {met: ‘false’})
MERGE (g)-[:isAnActivityStateOf]->(f)


MATCH (f:clearCarAccidentOntologyroot:clearCarAccident:activity),
(c:clearScene:activityState {met: ‘false’})
MERGE ©-[:isAnActivityStateOf]->(f)


MATCH (f:clearCarAccidentOntologyroot:clearCarAccident:activity),
(g:sceneClear:goal {met: ‘false’})
MERGE (g)-[:isAGoalOf]->(f)

MATCH (f:annotateScene:activityState {met: ‘false’}), (g:clearScene:activityState
{met: ‘false’})
MERGE (f)-[:isSucceededBy]->(g)

MATCH (f:clearScene:activityState {met: ‘false’}), (g:sceneClear:goal {met:
‘false’})
MERGE (f)-[:isSucceededBy]->(g)
```

```
CREATE (a:clearCarAccident:activity), (b:annotateScene:activityState {met:
'false'}), (c:clearScene:ActivityState {met: 'false'}), (d:sceneClear:goal {met:
'false', time_goal_created: '', time_goal_met: '', updateGoalWhen
'triggeredSituation.itemsOnScene 0'})
```

We now want to extend the existing person ontology for this situation. We will add the notion of medical records through multiple entities and add the concept of a person having them. Creating the records looks like this:

```
CREATE (a:HealthRecords:healthrecords:entity)-
[:containsBloodPressureRecords {situation_exists: 'accident'}]->
(b:BloodPressureRecords:bloodpressurerecords:entity)
```

And linking them to the general person ontology can be accomplished with the following stanza:

```
Match (d:Person:person:entity),(e:HealthRecords:healthrecords:entity)
CREATE (d)-[:containsHealthRecords {situation_exists: 'accident'}] ->(e)
```

Lastly, as this is a 2-car accident we will assume the default model has at least two drivers. We then ensure our model has 2 people in it and that each person we use is a copy of the PersonOntologyRoot by using the isAnInstantiationOf:

```
CREATE (y:Person1:person:entity),(z:Person2:person:entity)
```

```
MATCH (n:Person1),(m:PersonOntologyroot:Person:person:entity)
MERGE (n)-[:isAnInstantiationOf]->(m)
return m

MATCH (n:Person2),(m:PersonOntologyroot:Person:person:entity)
MERGE (n)-[:isAnInstantiationOf]->(m)
return m
```

Creating the vehicles is similar to the process required to create and use the person ontology. We will first create the ontology root for a vehicle. We then create a reference to a vehicle for each person. Finally, we link each vehicle to the two people that are involved in the car accident. Starting with the vehicle ontology:

```
CREATE (m:CarOntologyroot:Car:car:entity {vinNumber: '', licensePlate: '',
model: '', make: '', year: '', preCacheTag: 'Action: GET
http://localhost:8080/Rover3/tagQueryDatabase m.licensePlate m.isStolen'})
```

And then creating and linking to the carOntology for the individual cars as the car accident ontology as this is for a 2-car accident:

```
CREATE (s:Car1:car:entity),(t:Car2:car:entity)


MATCH (n:Car1),(m:CarOntologyroot:Car:car:entity)
MERGE (n)-[:isAnInstantiationOf]->(m)


MATCH (n:Car2),(m:CarOntologyroot:Car:car:entity)
```

```
MERGE (n)-[:isAnInstantiationOf]->(m)
```

Linking the cars to the individual drivers involved in the accident:

```
Match (m:Person1:person:entity),(n:Car1:car:entity)
MERGE (m)-[:wasDriving]-(n)

Match (m:Person2:person:entity),(n:Car2:car:entity)
MERGE (m)-[:wasDriving]-(n)
```

And finally associating these cars with the accident by saying they are involved in the two-car accident:

```
MATCH (n:Car1), (c:TwoCarAccident:CarAccident:Accident:situation)
MERGE (n)-[:isInvolvedIn]->(c)

MATCH (n:Car2), (c:TwoCarAccident:CarAccident:Accident:situation)
MERGE (n)-[:isInvolvedIn]->(c)
```

We want to also include the person who is acting as the first responder, an activity to help those injured and an arrest activity. All of these primitives might be required if one of the cars in the accident is listed as stolen or someone is found to be injured. We add a third person to include a support person in the relevant information set for this situation. This person will, by default, be instantiated when the car accident situation occurs. The activities are not auto-instantiated but a part of the model in the template store. The first stanza makes a first responder an added entity when the accident is instantiated:

```
CREATE (y:Person3:person:entity)

MATCH (y:Person3),(p:PersonOntologyroot:Person:person:entity)
MERGE (y)-[:isAnInstantiationOf]->(m)

MATCH (y:Person3),(m:CarAccident:accident:situation)
MERGE (y)-[:isAFirstResponderTo]->(m)
```

We then create an activity to make an arrest if one of the vehicles in the accident is stolen:

```
CREATE
(a:arrestPersonActivityOntologyroot:arrestPersonActivity:activity{ifEntityExists
As: ‘isStolen IS Yes’}), (b:arrestPerson:activityState {met: ‘false’, person: ‘’}),
(c:personArrested:goal {met: ‘false’, time_goal_created: ‘’, time_goal_met: ‘’})

MATCH (a:arrestPersonActivityOntologyroot:arrestPersonActivity:activity),
(b:arrestPerson:activityState {met: ‘false’, person: ‘’) MERGE (b)-
[:isAnActivityStateOf]->(a)

MATCH (a:arrestPersonActivityOntologyroot:arrestPersonActivity:activity),
(c:personArrested:goal {met: ‘false’, time_goal_created: ‘’, time_goal_met: ‘’})
MERGE (c)-[:isAnActivityStateOf]->(a)

MATCH (b:arrestPerson:activityState {met: ‘false’, person: ‘’),
(c:personArrested:goal {met: ‘false’, time_goal_created: ‘’, time_goal_met: ‘’})
MERGE (c)-[:isSucceededBy]->(b)
```

Finally, we add an activity to get medical support if someone is injured. Note that this activity would also be instantiated if a person in the context store was injured but there was no carAccident situation present in the context store:

CREATE (a:getMedicalHelpActivity:activity:ontologyRoot {ifSituationExists: ‘injuredPerson’, ifEntityExistsAs: ‘status CONTAINS injur’}), (b:callAmbulance:activityState {met: ‘false’, updateStateWhen ‘Ambulance.status called’}), (c:ambulanceArrives:ActivityState {met: ‘false’, updateStateWhen: ‘Ambulance.location onScene’}), (d:loadInjured:goal {met: ‘false’, time_goal_created: ‘’, time_goal_met: ‘’, updateGoalWhen: ‘triggeredEntity.location inAmbulance’}), (e:FirstResponder:person:entity {requiredEntity: ‘false’, linkNodeTo: ‘e providesMedicalSupportTo triggeredSituation fromNew’, linkedNodeTo: ‘e assists triggeredEntity’})

MATCH (f:getMedicalHelpActivity:activity {***ifSituationExists***: ‘injuredPerson’, ***ifEntityExistsAs***: ‘status CONTAINS injur’}), (e:FirstResponder:person:entity)
MERGE (e)-[:isAResponderWith]->(f)

MATCH (m:PersonOntologyroot:Person:person:entity {first_name:'', last_name:'', id:'', ssn:''}), (e:FirstResponder:person:entity)
MERGE (e)-[:isAnInstantiationOf]->(m)

MATCH (f:getMedicalHelpActivity:activity {ifSituationExists: ‘injuredPerson’, ifEntityExistsAs: ‘status CONTAINS injur’}), (g:callAmbulance:activityState {met: ‘false’})
MERGE (f)-[:isAnActivityStateOf]->(g)

MATCH (f:getMedicalHelpActivity:activity {ifSituationExists: ‘injuredPerson’, ifEntityExistsAs: ‘status CONTAINS injur’}), (h:ambulanceArrives:ActivityState {met: ‘false’})

MERGE (f)-[:isAnActivityStateOf]->(h)

MATCH (f:getMedicalHelpActivity:activity {ifSituationExists: ‘injuredPerson’, ifEntityExistsAs: ‘status CONTAINS injur’}), (i:loadInjured:goal {met: ‘false’, time_goal_created: ‘’, time_goal_met: ‘’})
MERGE (f)-[:isAnActivityStateOf]->(i)

MATCH (g:callAmbulance:activityState {met: ‘false’}), (h:ambulanceArrives:ActivityState {met: ‘false’}) MERGE (h)-[:isSucceededBy]->(g)

MATCH (h:ambulanceArrives:ActivityState {met: ‘false’}), (i:loadInjured:goal {met: ‘false’, time_goal_created: ‘’, time_goal_met: ‘’}) MERGE (i)-[:isSucceededBy]->(h)

8.2 Consuming the Ontology

Now that we have created all of the necessary primitives in the template store we are ready to consume the ontology for real-world examples. Once we have the ontology built we no longer deal in the world of Cypher and all interactions are handled through the web service through REST API calls. Using the REST URI commands outlined in section 6.5 we can instantiate models as situations occur and need to be represented, populate them with additional information as external entities observe them, track the progress of activities, and query any component of the context store. As we do that Rover3 will manage the relevance at runtime and work to ensure a relevant model is always present in the context store.

We begin by simply creating a situation. Continuing with the example above we will start by creating a two-car accident. The following URI will create a two-car accident in the context store:

http://localhost:8080/Rover3/createContext/Situation/twoCarAccident.

The output is:

```
Created new Situation with System id of 145
```

The Context Store now has a situation with an id of 145. Consumers can use this ID for updating, referencing, exploring, and linking to other objects in the Context Store as needed. Using the Web API, we can see all entities associated with the context. For example, we can pull the labels and ids for the entities in the situation with the following call:

http://localhost:8080/Rover3/getContext/Situation/65/entity/all

And the output for that will be:

```
[{"Person:person1":65},{"Person:person2":66},{"Car:car1":67},{"Car
:car2":68},{"HealthCareRecords:healthcarerecaords1":74},{"HealthCa
reRecords:healthcarerecaords1":75},{"BloodPresureRecords:bloodPres
sureRecord1":69},{" BloodPresureRecords:bloodPressureRecord2":71}]
```

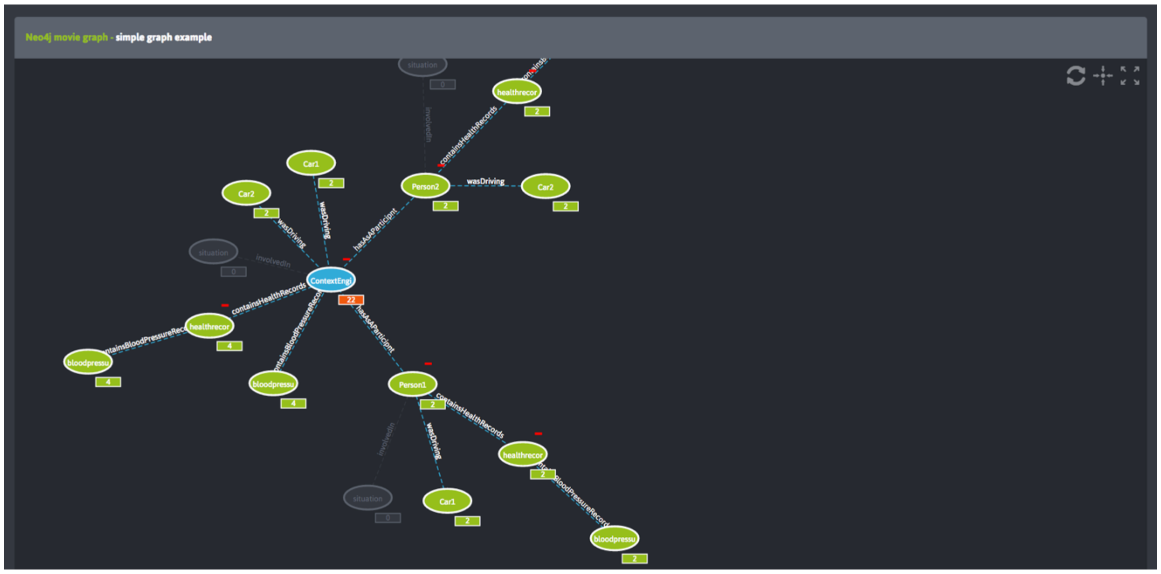
Using any of the ids that we get from the above output, we can further explore an entity. For example:

http://localhost:8080/Rover3/getContext/Entity/66/properties

The output will be:

```
{"properties(n)":{"last_name":"","id":"","model_id":"66"
,"first_name":"","ssn":""}}
```

We can also visually browse the context store. Prior to this the context store was empty. Going to the context browser and expanding on the tool we see the following in the context browser:



8.2.1 – The two-car accident situation in the context store after being instantiated through a createContext Web API call..

Now that the model exists in the context store we are ready to start placing discrete information about the entities and relations for the given situation into the context store. Using the ID and the various API calls, we can update the ssn, first, and last names accordingly:

http://localhost:8080/Rover3/updateContextByID/65/ssn/123456789

and the output is:

```
The ssn attribute was updated to 123456789 for primitive ID
65
Finished updating context...
```

http://localhost:8080/Rover3/updateContextByID/65/first_name/Nick

and the output is:

```
The first_name attribute was updated to Nick for primitive ID
65
Finished updating context...
```

http://localhost:8080/Rover3/updateContextByID/65/last_name/Gramsky

and the output is:

```
The last_name attribute was updated to Gramsky for primitive
ID 65
Finished updating context…
```

Querying that entity again we see the attributes are updated:

http://localhost:8080/Rover3/getContext/Entity/66/properties

and the output is:

```
{"properties(n)":{"last_name":"Gramsky,"model_id":"66"
,"first_name":"Nick","ssn":"123456789"}}
```

The instantiated model for the 2-car accident might not be enough to properly represent the actual situation at hand. Let us suppose the first car has a passenger and that passenger is injured. We can add the person and the update their status with the following WEB API calls:

http://localhost:8080/Rover3/createContext/Entity/Person/LinksTo/67/Via/wasAPassengerIn

and the output is:

```
Created new entity with System id of 78 Created new
relationship wasAPassengerIn with System id of 80 Linked
entity 79 -> entity 78 via relationship id 80.
Finished creating Context...
```

http://localhost:8080/Rover3/updateContextByID/78/status/injured

and the output is:

```
The status attribute was updated to injured for primitive ID
78
New Activity getMedicalHelp with ID 81 created and initiated
for ID 78
New Goal loadInjured with ID 82 created
Finished updating context...
```

Notice we created the 3rd person (the passenger of the first car) and linked it to the situation merely by creating a new relationship *wasAPassengerIn*. That relationship does not exist in the ontological store but is added to the context store. As the Rover3 framework does not require the ontological store to be the absolute authority on how entities can relate, external users can create a primitive with any label to properly reflect their surroundings. Despite the fact that a passenger did not exist as an option in the template store, when we set the status to *injured* an activity and a goal are created. The relevance for the situation has evolved and Rover3 automatically does that update. We now have 2 goals for the given situation and an activity is kicked off to get the injured person to a hospital.

Through template rules the act of updating context can instruct Rover3 to pre-cache additional information and kick off activities and/or expand goals. Police officers using this system for this situation would want to enter the license plate for the cars involved. In this case study, we enter a license plate, the status about it being stolen is pre-cached using the rules entered in the ontology, and an activity is kicked off to arrest the driver. Interfacing with the API and viewing the results looks like so:

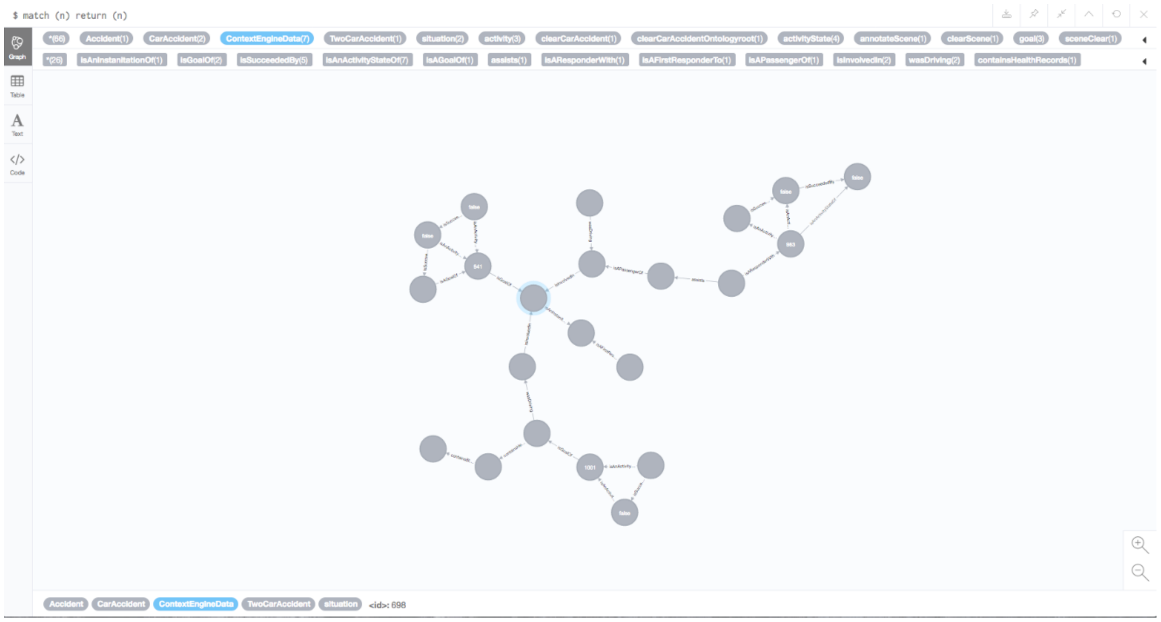
http://localhost:8080/Rover3/updateContextByID/68/licensePlate/ABC123

and the output is:

```
The licensePlate attribute was updated to injured for
primitive ID 68
Activity Manager is pre-caching the isStolen attribute for ID
68
The isStolen attribute is set to true
New Activity arrestPersonActivity with ID 88 created and
initiated for ID 78
New Goal personArrested with ID 89 created
Finished updating context...
```

We now have a new activity and goal simply by adding a value to an attribute for one of the cars in the instantiated model. Recall from section 7.1 when we built the activity *arrestPerson* activity we created a rule that kicked off that activity is any entity was labeled stolen. The pre-caching is what set the entity of the car to stolen and the constant relevance management of Rover3 is what allowed the new goals and activities.

While we have only added one discrete entity in this sub chapter (the passenger of car1), many other primitives were added through activities or supporting entities. Viewing all of the above changes in Neo4J's viewer we see the context store has expanded a bit.



8.2.1 – Updated Context Store after adding a passenger and enriching context of situation, resulting in added activities and entities.

We see a similar update through the web API:

http://localhost:8080/Rover3/getContext/Situation/145/entity/all

And the output for that will be:

```
[{"Person:person1":65},{"Person:person2":66},{"Car:car1":67},{"Car
:car2":68},{"HealthCareRecords:healthcarerecaords1":74},{"HealthCa
reRecords:healthcarerecaords1":75},{"BloodPresureRecords:bloodPres
sureRecord1":69},{"
BloodPresureRecords:bloodPressureRecord2":71},{“Person:person3”:78
},{“Person:firstResponder”:86}]
```

8.3 Achieving Goals with the Ontology

The work in section 8.2 instantiated and enriched the two-car accident situation. After gathering the initial information, the situation evolves from a single goal and single activity to a total of 3 goals and 3 activities. Consumers of the relevant information can use that guidance and information in the context store to help achieve those goals. Systems or users that directly interface with the Rover3 system can view the goals and activities through the web s   API. Our current list of activities and activity steps can be seen through the following call:

http://localhost:8080/Rover3/getContext/Situation/activityList

And the output will be:

```
[{"arrestPersonActivity":88,      met:false},{"getMedicalHelp":81,
met:false}, "clearAccident":85, met:false}]
```

http://localhost:8080/Rover3/getContext/Situation/goals

And the output will be:

```
[{"accidentClear":40,               met:false},{personArrested:89,
met:false},{"loadInjured":82, met:false}]
```

http://localhost:8080/Rover3/getContext/Activity/81/states

And the output will be:

```
[{state1:"callAmbulance",           met:false,            id:201},
{state2:"ambulanceArrives",          met:false,            id:202},
{state3:"loadInjured",      met:false,      id:203},      {goal:"
getPersonMedicalHelp", met:false, id:204}]
```

Knowing the goals and activities allows users to take the recommended actions to remedy the situation. With the 3 goals we have defined, we have varying levels of activities. The activities can either initiate actions for Rover3 to make or for consumers to enact. For example, Rover3 can take actions to call an ambulance through a web service call if a particular context were reached. This would be similar to the action we encoded in the car model where we pre-cached the *isStolen* attribute when the *licensePlate* attribute is populated in the context store.

Looking at what it takes to accomplish the goal *personArrested*, we look at the *arrestPerson* activity:

http://localhost:8080/Rover3/getContext/Activity/88/states

And the output will be:

```
[{state1:"arrestPerson”,            met:false,            id:301},
{goal:”personArrested”, met:false, id:302}]
```

We see we have an activity that has a single activity state and then a single goal. We can expect each state through the *getContext* call:

http://localhost:8080/Rover3/getContext/Activity/state/301

And the output will be:

```
[arrestPersonActivityState {state:1, met:false, person:’’}]
```

http://localhost:8080/Rover3/getContext/Activity/state/302

And the output will be:

```
[personArrested {state:goal, met:false, time_goal_created: ’12:49
01/18/2018’, time_goal_met: ‘’}]
```

Using the updateContext command we can set the states of each activity as they are accomplished. When all activity states are completed the goal for the activity also completes. Closing out the activity to arrest the person, we can simply set the met flag to true for the first state:

http://localhost:8080/Rover3/updateContextByID/301/met/true

And the output will be:

```
The met attribute was updated to true for primitive ID 301
The arrestPerson activity state has been set to met.
The goal arrest Person has been met and the activity arrestPerson
has completed.
Finished updating context...
```

We do not have to discretely accomplish each state of an activity. While the activity list is there to guide how to bets accomplish the goal, there might be external ambulance happened to be onsite already, one could simply add the ambulance to the situation. Using the *updateContext* API call we can link the ambulance to the situation and accomplish most of the getMedicalHelp activity with the following call:

http://localhost:8080/Rover3/createContext/Entity/Ambulance/LinksTo/Situation/145/AssistsWith

And the output will be:

```
Created new entity with System id of 111 Created new relationship
AssistsWith with System id of 112 Linked entity 111 -> situation 145
via relationship id 112.
Finished creating Context...
```

http://localhost:8080/Rover3/updateContextByID/111/location/onScene

And the output will be:

```
The location attribute was updated to onScene for primitive ID 111
The callAmbulance and ambulanceArrives activity states have been set
to met.
Finished updating context...
```

Looking at the activity states for the *getMedicalHelp* activity we now see the first two states have been met and all that remains it to load the injured. Using *updateContext* to set the goal to true, we can then follow-up and validate that the goals are met for the situation:

http://localhost:8080/Rover3/getContext/Activity/81/states

And the output will be:

```
[{state1:"callAmbulance",            met:true,            id:201},
{state2:”ambulanceArrives”,           met:true,           id:202},
{state3:”loadInjured”,  met:false,  id:203},  {goal:”personHelped”,
met:false, id:204}]
```

http://localhost:8080/Rover3/updateContextByID/203/met/true

And the output will be:

```
The met attribute was updated to true for primitive ID 203
The  goal  getPersonMedicalHelp  has  been  met  and  the  activity
getMedicalHelp has completed.
Finished updating context...
```

http://localhost:8080/Rover3/getContext/Activity/81/states

And the output will be:

```
[{state1:"callAmbulance",            met:true,            id:201},
{state2:”ambulanceArrives”,           met:true,           id:202},
{state3:”loadInjured”,      met:true,      id:203},      {goal:”
getPersonMedicalHelp”, met:true, id:204}]
```

Following the same convention with the activity created with the situation itself, we can eventually bring the situation to a close. We omit that last bit of activity as it is merely a repeat of the last two activities. We can continue to add information to the situation. The situation is no longer active (the active flag is no longer set and the situation handler archives the situation) but this is kept in the context store as it represents a collection of discrete primitives that existed in the real world at one time.

9 Extensibility Study

So far, we have seen ADIM and the Rover3 system utilized through an emergency response scenario. Let us now explore the effort to add another domain or scenario to our ontological store. In doing so, we will compare the effort to do the same exercise for the existing Rover2 system.

Using the ontological models in chapter 8 as the basis for expansion, let us loosely explore what would be required. We will briefly discuss what is required for each step. For each evaluation, we will consider expanding or adding onto existing models as well as creating new primitives. Any situation needs new situation primitives added as situations are what is needed to derive context. We will also look at new activities, entities, and triggers. We will omit goals (as they are necessary with situations and activities and must also be created) and relationships as they do not impact this study. As we explore the efforts to create a new domain in the proceeding sections, we will consider the supporting primitives for adding the situation 'interview'.

9.1 Extending Existing Models in Rover3

Re-using existing models is relatively easily as one can simply append to the existing ontological store as needed. The process is the same one would adopt when initially creating primitives, one simply writes Cypher statements to load into the ontological store. An example here would be adding professional certifications to the Person entity for an interview situation. This can be seen as:

```
CREATE (a:ProfessionalRecords:professionrecords:entity)-
[:containsCertifications ] -> (b:Certifications:certifications:entity)

MATCH (d:Person:person:entity),(e:ProfessionalRecords:entity)

CREATE (d)-[:contains {situation_exists: ‘interview’}]->(e)
```

When adding a new situation, we simply create new ones. As all this requires are new Cypher entries, this is rather straight forward. An example would be:

```
CREATE (g:InterviewOntologyroot:Interview:accident:situation {location:'',
newActivity:'conductInterview'})
```

New activities or triggers require the same effort as we can do so through new Cypher commands. The same holds true for modifying activities or triggers. When creating new situations or entities, we can modify existing activities based on the new primitives. An example of modifying an existing activity upon creating a new situation would be:

```
MATCH (a:arrestPersonActivityOntologyroot:arrestPersonActivity:activity)
SET a.ifSituationExists = ‘assultedInterviewer’
```

Similarly, instantiating new entities at runtime or having new situation instantiate existing or new entities is as simple as writing or modifying the ontological store through Cypher commands; we add triggers the same way we did in the previous chapter.

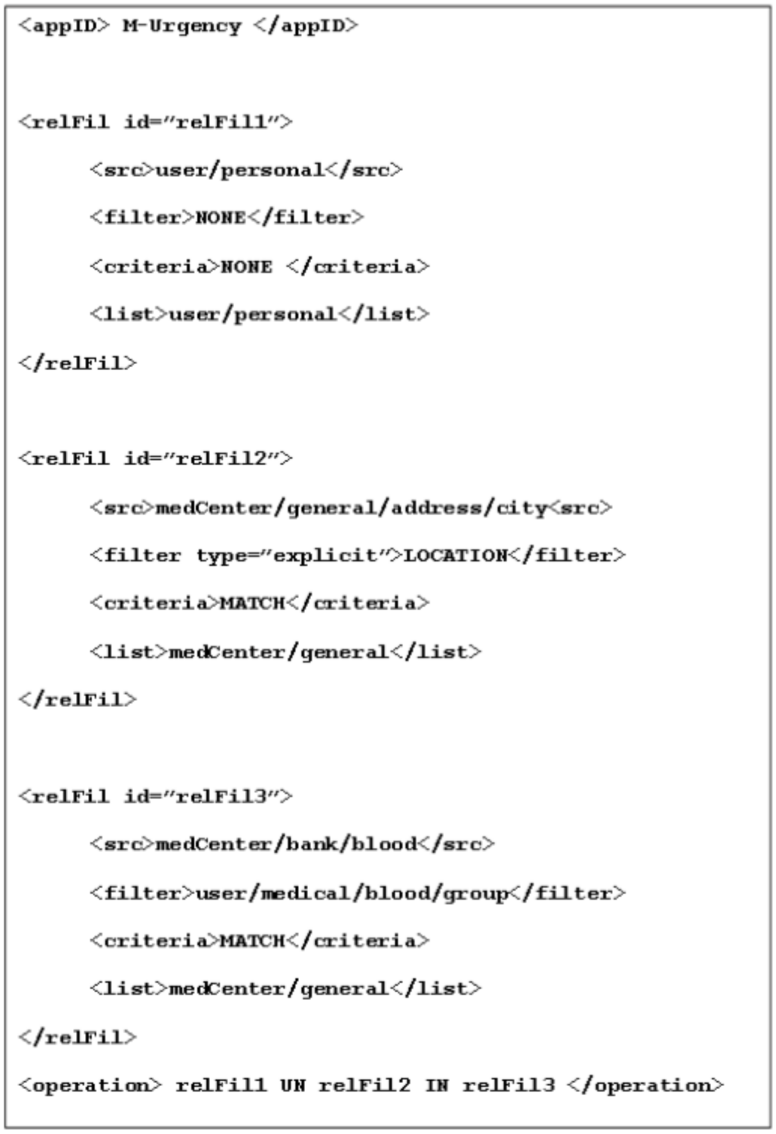
9.2 Extending Existing Models in Rover2

Expanding the ontological store in Rover2 [14] can require a little bit more effort. If we are simply adding new primitives to the ontological store, the effort is the same as if one were creating new primitives. Rover2 uses the OWL ontological framework, so we simply draft new primitives in XML using the OWL language. Figure 9.2.1 shows an example entity in Rover2. Adding to this model (either creating new primitives or adding attributes to the existing one) is as simple as drafting and modeling the primitive in question or expanding the OWL object as needed.

```
⟨owl:NamedIndividual rdf:about="#determineEvacuationTime"⟩
    ⟨rdf:type rdf:resource="&goal;goal"/⟩
    ⟨rdfs:label xml:lang="en"⟩determineEvacuationTime⟨/rdfs:label⟩
    ⟨goal:hasDescription rdf:datatype="&xsd;dateTime"⟩Determine time for evacuation⟨/goal:hasDescription⟩
⟨/owl:NamedIndividual⟩
⟨owl:NamedIndividual rdf:about="#getTempReadings"⟩
    ⟨rdf:type rdf:resource="http://mind7.cs.umd.edu:8134/Rover/activity#activity"/⟩
    ⟨rdfs:label xml:lang="en"⟩
getTempReadings⟨/rdfs:label⟩
    ⟨time rdf:resource="#getTempReadingsTime"/⟩
⟨/owl:NamedIndividual⟩
⟨owl:NamedIndividual rdf:about="#getTempReadingsTime"⟩
    ⟨rdf:type rdf:resource="&time;time"/⟩
    ⟨rdfs:label xml:lang="en"⟨getTempReadingsTime⟩/rdfs:label⟩
    ⟨time:startTime rdf:datatype="&xsd;dateTime"⟨2012-09-21T14:32:12⟩/time:startTime⟩
⟨/owl:NamedIndividual⟩
```

Figure 9.2.1 – Sample primitive in OWL notation from Rover2

New situations, however, require more than just creating a new model. Rover2, much like Rover3, uses situations as the basis for defining context. Filters are created for given situations or applications (application can have assumed situations or a subset of situations that are inferred). Figure 9.2.2 is an example of such a filter. These filters, however, are rather static in nature and context is static for any given situation. That is, every time the same situation is instantiated, the same contextual model is delivered. If additional information is learned and the situation evolves, end users must self-manage the relevant context. Any automatic expansion or of the instantiated context is only accomplished through additional, external software. In order to do this at runtime, the external software must monitor the information store within the Rover2 system and take necessary blocking actions on each update.



```
<appID> M-Urgency </appID>


<relFil id="relFil1">
      <src>user/personal</src>
      <filter>NONE</filter>
      <criteria>NONE </criteria>
      <list>user/personal</list>
</relFil>


<relFil id="relFil2">
      <src>medCenter/general/address/city<src>
      <filter type="explicit">LOCATION</filter>
      <criteria>MATCH</criteria>
      <list>medCenter/general</list>
</relFil>


<relFil id="relFil3">
      <src>medCenter/bank/blood</src>
      <filter>user/medical/blood/group</filter>
      <criteria>MATCH</criteria>
      <list>medCenter/general</list>
</relFil>
<operation> relFil1 UN relFil2 IN relFil3 </operation>
```

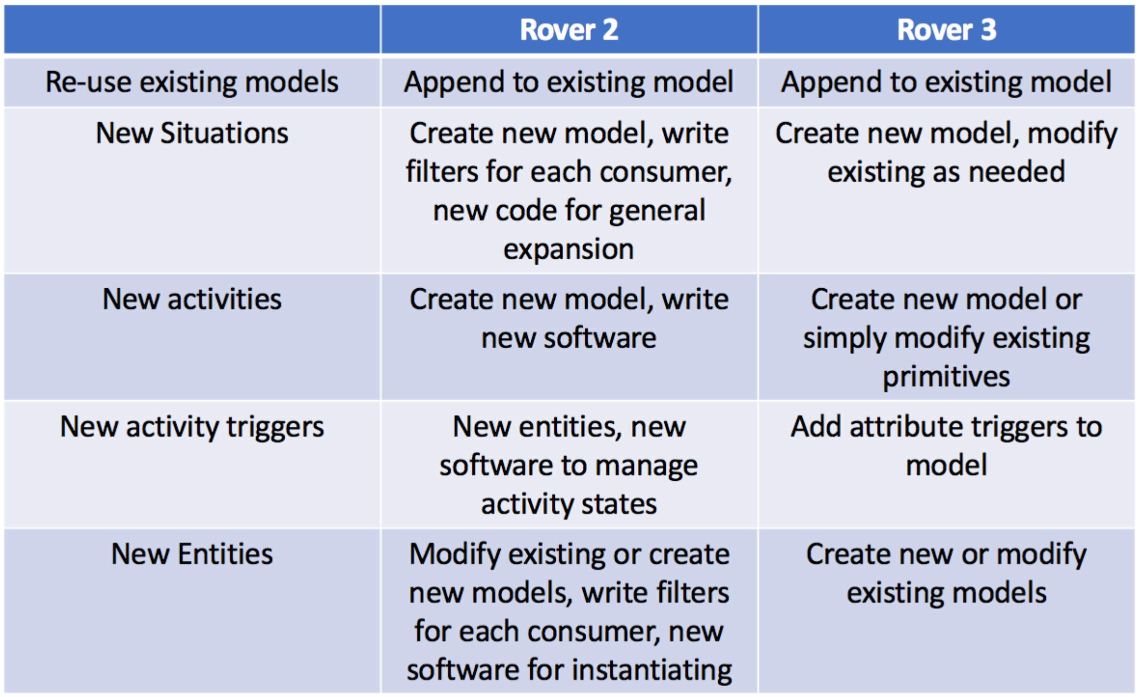
Figure 9.2.2 – Sample context filter used in Rover2

New activities or triggers are where the effort to manage context starts to increase drastically. Rather than simply append to existing ontological store and create new filters, users of the system are forced to write external software to manage the efforts. Rover2 does not have an activity manager that progresses through the various states of a given activity. Though activities had goals, there is no way to encode the various steps and systematically track progress to the goal without external software. Instantiation too was only accomplished through external software, meaning we could not trigger the existence of an activity at runtime based on the ontological model itself. The result is multiple pieces of software have to be written for each activity *after* we create the ontological model for the activity itself.

Instantiating new entities carries a similar tax and scaling issues. Though we were able to extend the ontological store quite easily as outlined in the beginning of this section (simply creating new OWL objects or extending existing one), instantiating them requires one, if not two, additional steps. Each time an entity is appended, all filters that contain that entity must be updated. If a new entity is created, all filters need to be evaluated if the entity is relevant to the situation or application. As the Rover2 application was centered around context and not auto-instantiation, users are required to write new software to instantiate new entities.

9.3 Comparison of the Two Systems

Evaluating the two systems on a case by case basis, we see the effort in each scenario is more favorable for Rover3 than it is for Rover2. Figure 9.3.1 summarizes the different scenarios. In summary, Rover3 allows for the addition of new domains and collections of primitives without external software. Ontology authors just need to model their domain, defining the structure of the information that represents the entities of that space as well as the rules for context manipulation at runtime. Rover2, however, requires external software to be written or modified in most scenarios in order to break free from rigid and very discrete context as situations evolve.



	Rover 2	Rover 3
Re-use existing models	Append to existing model	Append to existing model
New Situations	Create new model, write filters for each consumer, new code for general expansion	Create new model, modify existing as needed
New activities	Create new model, write new software	Create new model or simply modify existing primitives
New activity triggers	New entities, new software to manage activity states	Add attribute triggers to model
New Entities	Modify existing or create new models, write filters for each consumer, new software for instantiating	Create new or modify existing models

Figure 9.3.1 – Table summarizing the impacts of adding new primitives to the ontological store in Rover2 and Rover3

10 Contribution and Future Work

We have explored new ways to manage and deliver context to end users. Explaining what a context aware system is and the principals of how one should be built, we explored how existing systems and approaches had more to offer. Seeking to deliver relevant context, we defined the basic building blocks necessary when delivering such information. Leveraging our proposed design principles, we presented how a system should deliver and evolve context at run time. Such systems continually present and update consumers with the information deemed relevant for any given situation. ADIM was presented as a methodology to encode relevance in ontological models. This modeling, when used with a proper context management system can manage run time evolution of context. We built Rover3 as an extensible framework with this purpose in mind. We showcased how this could work with our case study of an emergency response situation. We closed with a comparison of Rover3 and Rover2 and showed how consumers of Rover3 need not write any additional software to instantiate and manage relevant context.

10.1 Contribution

In this dissertation, our contributions have been the following:

- Defining what it means to model any situation. By exploring the various components needed for context modelling, we provide all of the necessary components to do so in a way that best represents reality.
- Defining and designing how a system that delivers situationally relevant context at run-time should be engineered. We do so by outlining what it means for context to expand at run time, how to ensure such a system

delivers relevant context to end consumers, and how such a system should be designed.

- Introducing, implementing, and utilizing the Automatic Dynamic Information Model (ADIM) methodology as a way to encode relevance within an information model. Doing so enables context aware systems to deliver relevant information without additional software.
- Designing, implementing, and deploying the Rover3 context aware system, leveraging the design tenets of context aware systems. Using Rover3 and ADIM, we walkthrough a case study of developing information models for and delivering context within the emergency services domain. An extensibility study is performed to show how effortless it is to use Rover3 and ADIM over Rover2.

10.2 Future Work

While we have made several contributions in context management and context aware systems, our work outlined in this dissertation could greatly benefit from follow-on work. Opportunities to significantly lower the friction to create and edit both ontological and instantiated context models, additional levels of fidelity to define relevance, and additional ontological primitives all could further strengthen this area of research.

The methods for creating, editing, and viewing context models could benefit from substantial gains in usability. We chose to simply rely on Cypher as the method for encoding ADIM and context in our ontological store simply as it did not require any additional work. Developing an easier way to input and manage ontological models would further lower the barrier for SMEs to use the system. Doing so should look to better align with the *Minimize Abstraction of Reality* tenet as the existing modeling method requires a bit of effort. Once models are instantiated in the context store, simply clicking on and editing primitives would be much easier than individual web service calls. The current method requires context switching between viewing a model in the context store and sending individual POST calls to the controller. Moving this action to the GUI represented entity is the next logical step for managing instantiated and existing context. Lastly, color-coding goals and actions would help consumers of the system quickly identify what state the various primitives are in. Currently one has to inspect each object through web calls. This is another example of context switching, this time to observe current values.

The relationships and variability in the ontological model have opportunities to be more than binary in nature. Throughout our contribution, any relationship, attribute, ADIM action/trigger, or ontology were binary in nature – some criteria defined if something were to be considered relevant (and thus instantiated in the context store) or not.

Bibliography

[1] K. Henrickson and Jawiga Indulska, "Modeling context information in pervasive computing systems.," in *LNCS 2414: Proceedings of 1st International Confer ence on Pervasive Computing*, Berlin / Heidelberg, 2002.

[2] F. J. Riggins and S. F. Wamba, "Research Directions on the Adoption, Usage, and Impact of the Internet of Things through the Use of Big Data Analytics.," in *48th Hawaii International Conference on System Sciences (HICSS)*, 2015.

[3] A. Jamies, N. Sebe and D. Gatica-Perez, "Human-centered computing: a multimedia perspective.," in *Proceedings of the 14th annual ACM international conference on Multimedia.*, Santa Barbara, 2006.

[4] M. Baldauf, S. Dustdar and F. Rosenburg, "A survey on context-aware systems," *Int. J. Ad Hoc and Ubiquitous Computing,* vol. 2, no. 4, pp. 263-277, 2007.

[5] K. Pathan, S. Reiff-Marganiec and N. Channa, "Reaching activities by places in the context-aware environments using software sensors.," *Journal of Emerging Trends in Computing and Information Sciences,* vol. 2, no. 12, 2011.

[6] H. Yin, B. Cui, L. Chen, Z. Hu and C. Zhang, "Modeling location-based user rating profiles for personalized recommendation.," *ACM Transactions on Knowledge Discovery from Data (TKDD),* vol. 9, no. 3, p. 19, 13 April 2015.

[7] P. Bhargava, N. Gramsky and A. Agrawala, "SenseMe: a system for continuous, on-device, and multi-dimensional context and activity recognition.," in *Proceedings of the 11th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2014.

[8] H. Chen, T. Finin and A. Joshi, "n Ontology for Context-Aware Pervasive Computing Environments," Cambridge University Press, Baltimore, 2003.

[9] W. Woerndl, C. Schueller and R. Wojtech, "A hybrid recommender system for context-aware recommendations of mobile applications."," in *Data Engineering Workshop, 2007 IEEE 23rd International Conference on*, 2007.

[10] M. V. Setten, S. Pokraev and J. Koolwaaij, "Context-aware recommendations in the mobile tourist application COMPASS.," in *International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems.*, 2004.

[11] S. Yang, A. Huang, R. Chen and S.-S. Tseng, "Context Model and Context Acquisition for Ubiquitous Content Access in ULearning Environments," in *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06)* , 2006Taichung.

[12] C. B. Almazan, "Rover: Architectural Support for Exposing and Using Context," College Park, 2010.

[13] C. Almazan, M. Youssef and A. Agrawala, "Rover: An integration and fusion platform to enhance situational awareness.," in *007 IEEE International Performance, Computing, and Communications Conference*, 2007.

[14] S. Krishnamoorthy, "ROVER-II: A CONTEXT-AWARE MIDDLEWARE FOR PERVASIVE COMPUTING ENVIRONMENTS," University of Maryland, College Park, 2012.

[15] P. Bhargava, S. Krishnamoorthy and A. Agrawala, "An ontological context model for representing a situation and the design of an intelligent context-aware middleware," in *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, 2012.

[16] P. Bhargava, S. Krishnamoorthy and A. Agrawala, "RoCoMo: a generic ontology for context modeling, representation and reasoning in a context-aware middleware," in *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, 2012.

[17] S. Krishnamoorthy and A. Agrawala, "M-Urgency: a next generation, context-aware public safety application," in *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, 2011.

[18] S. Krishnamoorthy, "Context-aware, technology enabled social contribution for public safety using M-Urgency," in *Proceedings of the 14th international conference on Human-computer interaction with mobile devices and services*, 2012.

[19] S. Krishnamoorthy and A. Agrawala, "M-Urgency: a next generation, context-aware public safety application," in *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, 2011.

[20] M. Miraoui, S. El-etriby, C. Tadj and A. Zaid Abid, "Ontology-Based Context Modeling for a Smart Living Room," in *Proceedings of the World Congress on Engineering and Computer Science* , San Francisco, 2015.

[21] N. Gramsky and H. Samet, "Seeder finder: identifying additional needles in the Twitter haystack.," in *Proceedings of the 6th ACM SIGSPATIAL International Workshop on Location-Based Social Networks*, Orlando, 2013.

[22] Y. Zou, T. Finin and H. Chen, "F-owl: An inference engine for semantic web.," in *nternational Workshop on Formal Approaches to Agent-Based Systems*, Berlin, 2004.

[23] R. Gove, N. Gramsky, R. Kirby, E. Sefer, A. Sopan, C. Dunne, B. Shneiderman and M. Taieb-Maimon, "NetVisia: Heat map & matrix visualization of dynamic social network statistics & content.," in *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom)*, Boston, 2011.

[24] W. J. Clancey, Situated Cognition, Cambridge: Cambridge University Press.

[25] H. Yin, B. Cui, L. Chen, Z. Hu and Z. Huang, "A temporal context-aware model for user behavior modeling in social media systems.," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014.

[26] M. Uschold and M. Gruninger, "Ontologies: Principles, methods and applications," *Knowledge Engineering Review,* vol. 11, no. 2, pp. 93-155, 1996.

[27] A. Dey, "Understanding and Using Context," in *Personal and ubiquitous computing*.

[28] N. A. Bradley and M. Dunlop, "Towards a Multidisciplinary Model of Context to Support Context-Aware Computing," *HUMAN – COMPUTER INTERACTION,,* vol. 20, pp. 403-446, 2005.

[29] G. D. Abowd, "owards a better understanding of context and context-awareness.," in *nternational Symposium on Handheld and Ubiquitous Computing*, Berlin, 1999.

[30] P. Bellavista, A. Corradi and M. Fanelli, "A survey of context data distribution for mobile ubiquitous systems.," *ACM Computing Surveys (CSUR),* vol. 44, no. 4, p. 24, 1 August 2012.

[31] F. Azouaou and C. Desmoulins, " Using and modeling context with ontology in e-learning: the case of teac her's personal annotation," in *Hndbook of Research on Interactive Information Quality In Expanding Social Communications*, Hershey, PA: IGI Global, 2014, p. 435.

[32] H. Guermah, T. Fissa, H. Hafiddi, M. Nassar and A. Kriouile, "An Ontology Oriented Architecture for Context Aware Services Adaptation," *International Journal of Computer Science Issues,* vol. 11, no. 2, p. 10, 2014.

[33] "Ontology," [Online]. Available: https://en.wikipedia.org/wiki/Ontology. [Accessed 8 May 2016].

[34] Z. Ming, S.-x. Li and X.-r. Fang, "Research on Ontology-Oriented Domain Analysis on MIS," in *Formal Methods and Software Engineering 4th International Conference on Formal Engineering Methods*, Shanghai, China, 2002.

[35] S. Krishnamoorthy, P. Bhargava, M. Mah and A. Agrawala, "Representing and Managing the Context of a Situation," *The Computer Journal,* vol. 55, no. 8, pp. 1005-1019, 2011.

[36] A. Zimmermann, A. Lorenz and R. Oppermann, "An operational definition of context," in *International and Interdisciplinary Conference on Modeling and Using Context*, Berlin, 2007.

[37] P. Makris, D. Skoutas and C. Skianis, "A Survey on Context-Aware Mobile and Wireless Networking: On networking and computing environments' integration," *IEEE COMMUNICATIONS SURVEYS & TUTORIALS,* vol. 15, no. 1, pp. 362-386, 2013.

[38] C. Perera, A. Zaslavsky, P. Christen and D. Georgakopoulos, "Context Aware Computing for The Internet of Things: A Survey," *Communications Surveys & Tutorials,* vol. 16, no. 1, pp. 414-454, 2014.

[39] A. Dey and G. Abowd, "CybreMinder: A Context-Aware System for Supporting Reminders," *Handheld and Ubiquitous Computing,* pp. 172-186, 2000.

[40] M. Frejus, M. Dominici, F. Weis, G. Poizat, J. Guibourdenche and B. Pietropaoli, "Changing Interactions to Reduce Energy Consumption: from Activity Analysis to the Specification of a Context-Aware System," in *CHI '13*, Paris, France, 2013.

[41] K. Rehman, F. Stajano and G. Coulouris, "Architecture for Interactive Context-Aware Applications," *Pervasive Computing,* vol. 6, no. 1, pp. 73-80, 2005.

[42] Y. Zhang, S. Song and P. Tan, "A whole-room 3D context model for panoramic scene understanding," in *European Conference on Computer Vision*, Berlin, 2014.

[43] C. Mettouris and G. Papadopoulos, "Contextual modelling in context-aware recommender systems: a generic approach," in *Web Information Systems Engineering–WISE 2011 and 2012 Workshops*, Berlin, 2013.

[44] S. Yau and J. Liu, "Hierarchical situation modeling and reasoning for pervasive computing," in *he Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the Second International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA'06)*, 2006.

[45] N. Baumgartner and W. Retschitzeggar, "A survey of upper ontologies for situation awareness," in *Proc. of the 4th IASTED International Conference on Knowledge Sharing and Collaborative Engineering, St. Thomas, US VI*, 2006.

[46] Y. Jung, J. Ryu, K.-m. Kim and S.-H. Myaeng, "Automatic construction of a large-scale situation ontology by mining how-to instructions from the web," *SciendDirect,* vol. 8, no. 2, 18 April 2010.

[47] R. Hoffman, "Human-centered Computing Principles for Advanced Decision Architectures," Army Research Laboratory, 2004.

[48] J. Ye, L. Coyle, S. Dobson and P. Nixon, "Ontology-based models in pervasive computing systems," *The Knowledge Engineering Review,* vol. 22, no. 04, pp. 315-347, 2007.

[49] "Intro To Cypher," Neo4J, 22 May 2016. [Online]. Available: http://neo4j.com/developer/cypher-query-language/. [Accessed 22 May 2016].

[50] P. Hayes, "RDF Semantics," 2004. [Online]. Available: http://www.w3.org/TR/2004/ REC-rdf-mt-20040210/. [Accessed 22 May 2016].

[51] M. Dean and RV Guha, "OWL Web Ontology Language," 2004. [Online]. Available: https://www.w3.org/TR/2004/REC-owl-ref-20040210/. [Accessed 22 May 2016].

[52] X. H. Wang, D. Q. Zhang, T. Gu and H. K. Pung, "Ontology based context modeling and reasoning using OWL," in *Pervasive Computing and Communications Workshops*, 2004.

[53] M. Niepert, C. Buckner and C. Allen, "A dynamic ontology for a dynamic reference work," in *Proceedings of the 7th ACM/IEEE-CS joint conference on Digital libraries*, 2007.

[54] F. Zablith, "Dynamic ontology evolution," 2008. [Online]. Available: http://oro.open.ac.uk/23527/.

[55] "Popoto.js," [Online]. Available: http://www.popotojs.com/. [Accessed 12 December 2016].

[56] Neo4J, " Neo4j Browser User Interface Guide," Ne4J, [Online]. Available: https://neo4j.com/developer/guide-neo4j-browser/. [Accessed 1 Oct 2017].