2014-04-16

Adding an Ontology Filter to the Knowledge Base in CASA

Singh, Baljeet

UNIVERSITY OF CALGARY

Adding an Ontology Filter to the Knowledge Base in CASA

by

Baljeet Singh

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTERS OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

APRIL, 2014

Abstract

Ontologies are being used in a lot of applications today. Ontologies provide extensive knowledge about a domain. Knowledge bases are also being used in a lot of software applications and agent based applications. Agent based systems are used in various applications ranging from biomedical applications to financial applications and from military applications to graphic applications.

This thesis presents the integration of the ontologies and the knowledge base in an agent based system called Collaborative Agent System Architecture (CASA). CASA uses a knowledge base from another agent based system called JAVA Agent Development Environment (JADE) and uses the OWL2 ontology engine. In CASA, agents are queried without using ontological knowledge, only using knowledge from the underlying knowledge base. This gives us poor quality results. The ontology filter is then added to the system. The agents are then queried with the knowledge from both the ontologies and the knowledge base (using ontology filter). The later shows improved results (more accurate results). This integration of the ontologies and the knowledge base could be extremely useful in a lot of agent based applications.

Acknowledgments

I wish to thank following people who encouraged and helped me at each and every point of this thesis. I would like to extend my sincere gratitute to them.

Thank you to my family (mom, dad, sister, uncle and aunt) for all the love, encouragement and support. I could not have done this without your support.

Thank you to my supervisor and mentor Dr. Robert Carl Kremer for providing me with this wonderful opportunity to come to Calgary and work under your guidance. Your wisdom and encouragement has guided me in maturing academically, professionally, and personally. Thanks for your limitless support, feedback, and for being a source of inspiration during the course of this thesis.

Thanks to all the people of the Artificial Intelligence lab for being a part of this thesis.

Finally, I would like to thank my friend Amandeep Kahlon for going through the process of graduate school with me, for sharing the little successes and failures of the process, and for always being there for me. Doing this work without your support would have been infinitely harder.

Table of Contents

List of Tables

List of Figures and Illustrations

Chapter 1

Introduction

1.1 Background

Getting good search results for a query has always been of great interest to researchers. Knowledge based systems use knowledge bases to give results for the user queries [8]. Early knowledge bases acted like databases and only syntactic matching was used to give results [9]. With the advancements in technology, this is not sufficient. Knowledge bases have become more powerful now. The results of the user queries should be generated using semantic matching: ontologies on top of syntactic matching [9].

1.1.1 Knowledge Base and Ontologies

A knowledge base stores facts believed by the agents according to some specific applications. A knowledge base stores raw facts without knowing their semantic meaning [10]. For instance, if an agent is informed that `(hasfur mammal)` is true, then it answers true if we query `query-if (hasfur mammal)`. The query is asking if mammals have fur. But if we query `query-if (hasfur dog)`, asking if dogs have fur, the result in this case is false (nil). Clearly, this result is wrong because a dog is a mammal, an ontological fact.

"Ontology is the term used to refer to the shared understanding of some domain of interest which may be used as a unifying framework [11, p.4]." An ontology incorporates a type lattice in itself. Each entity of the lattice is termed a type. For instance, `(declType "cat")` declares a type `cat` in the ontology. These types have certain relations defined among themselves. Using of these relationships on top of the knowledge from a knowledge base could really contribute to produce better query results. An example ontology lattice is

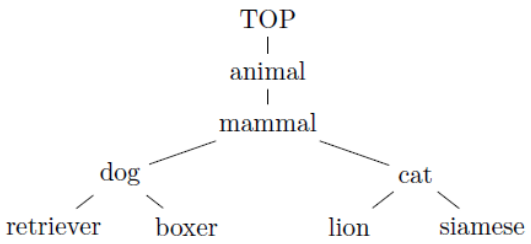

Figure 1.1: Example Ontology lattice[1]

given in Figure 1.1.

1.1.2 Agent Based Systems

An agent based system is composed of multiple agents which interact to solve a given problem or to achieve a goal [12]. This research only deals with the software agents. These agents interact within an environment. An agent based system is used in areas like artificial intelligence, distributed systems, robotics, game theory, psychology etc [12]. Agent based systems have certain goals to achieve. Some general goals include higher efficiency, faster results and improved solutions [12]. Some other goals include easily adaptable, extend-able and reliable [12]. This research focuses on improving answers to a query in agent based system.

1.1.3 Collaborative Agent Systems Architecture

Collaborative Agent Systems Architecture (CASA) is a platform for developing agent based systems [1]. One major difference between the CASA and the other platforms is that CASA can use different agent models for conversations among its agents. CASA can supports Foundation for Intelligent Physical Agents (FIPA) [13] as well as others standards . CASA has a knowledge base; which it has acquired from Java agent development enviornment (JADE) semantic knowledge base. JADE is another agent development environment which follows the FIPA standards [13] strictly.

Every agent in CASA has its own knowledge base. A knowledge base is like a database which has all the knowledge of a particular agent. An agent uses this knowledge to produce the results for queries. When a query is sent to an agent, it is passed to the knowledge base for searching. If the knowledge base has direct knowledge about the query then true is produced as the result, otherwise false is returned as the result. So, the JADE semantic extension knowledge base does not have any ontological reasoning while producing the results for the query.

This thesis discusses the effect of adding ontology filter on top of the knowledge base. This chapter aims at providing an introduction to the thesis. Section 1.1 provides an overview of agent based systems, knowledge bases and ontologies. Section 1.2 and 1.3 discusses the motivation behind the thesis and research questions, respectively. Sections 1.4, 1.5 and 1.6 focus on research goals, aim and objectives of this research and provides an overview of the thesis structure, respectively.

1.2 Motivation

Collaborative Agent Systems Architecture (CASA) is being used in the *Artificial Intelligence lab at University of Calgary*. CASA inherits a knowledge base from Java agent development enviornment (JADE) semantic extension knowledge base. The agents make use of the knowledge from the JADE semantic knowledge base to answer user queries. The results were only true for direct matches. This means, if a queried fact is exactly present in KB, only then true is returned as the result otherwise false is returned as the result. Ontologies are also part of the CASA system. CASA uses the OWL2 ontology engine. The ontologies will be explained in detail in Chapter 2. The queries to the OWL2 ontology were also not rich enough to give the kind of results required by an agent based system. So both these capabilities should be integrated to get reasonable results for agent based applications. The primary goal of the

research is to integrate the ontologies and the knowledge from the knowledge base. Therefore, the intent of this research will be to make the queries go through an ontology filter and produce more appropriate results. Specially, the focus is on two aspects:

- To make results for a query more realistic and accurate.
- To make searching as efficient as possible.

This is the primary driving force for this thesis.

1.3 Research Questions

A study about ontologies and knowledge bases forms the foundation of this thesis. Exploratory research is employed to gather insights about the ontologies and the knowledge bases. Exploratory studies use secondary research methods like literature reviews and research papers. The scope of the problem is explored in detail in these studies.

Three broad research questions were identified -

- How are queries answered by the agents in CASA's (JADE Semantic Extension's) knowledge base? This question deals with determining the process employed in CASA to answer user queries. This is an important question because after knowing the current process, new filters could be added to this process.
- How can the ontology filter be applied on top of the knowledge base? This becomes the next question of interest. The ontology filter will be added to the CASA system. Queries will be passed through the ontology filter. This process will be tested against different test cases.
- How efficient and accurate are the results from this new process? Accuracy here means to determine if the results produced are correct and efficiency means to determine the time taken for producing the result.

1.4 Aims and objectives

The main aim of this research is to improve the usefulness of the CASA (JADE semantic extension) knowledge base by combining its capabilities with those of CASA's (OWL2) ontological capabilities. The main objectives of this thesis for fulfilling the above aim are:

1. Explain the CASA knowledge base (JADE semantic knowledge base) and its use by agents for answering user queries. This means understanding the knowledge base structure and current process of producing the results for queries. This also includes understanding the way ontologies are structured and used in the CASA system.

2. Extending the current process of producing the results for queries, by developing an ontology filter for the CASA system. This filter will work on top of CASA's knowledge base and add ontological knowledge on top of the knowledge from the knowledge base.

3. The ontology filter will be tested for different queries and the results will be analyzed. This knowledge from ontologies will improve the overall results of the queries and make the knowledge base more useful in agent based applications.

1.5 Thesis Overview

This chapter details the background for this thesis; it also includes a description of agent based systems. After that, research questions and goals are discussed. Chapter 2 presents the background and related work about the ontologies and the knowledge bases and other related works. Chapter 3 presents the requirements for the ontology filter implementation. Chapter 4 presents the details of the design and implementation of the ontology filter. Chapter 5 presents the details of the testing results of the ontology filter. The different test cases are

detailed and the results are analyzed. Chapter 6 presents the contributions of the research and future directions in this area.

Chapter 2

Background

This chapter aims at providing a contextual background for this research. In recent times work done (based on FIPA) on ontologies and knowledge bases has been application oriented. The research has not focussed on the inner details. So, a lot of literature cited in this thesis is from 90's. This is also the reason for lot of citations from websites in this thesis.

Section 2.1 focuses on agent based systems, describing three such systems: JADE [14] , Cougaar [15] and CASA [1]. Section 2.1.4 compares the three systems. Section 2.2 gives insight into the logics in computer science and describes predicate and description logics. Section 2.3 focuses on the ontologies, describing their meaning and uses. Section 2.3.1.2 gives an explanation about OWL ontologies and Section 2.3.2 gives OWL2 relation to description logics. OWL ontologies are used in this research work. Section 2.4 describes knowledge bases in general, as well as various ways to represent the knowledge in knowledge based systems.

2.1 Agent Based Systems

Agent based systems are composed of multiple agents which interact to solve a given problem or to achieve a goal [12]. An agent is an autonomous entity. Agents which use knowledge to take decisions are termed as intelligent agents [12]. Software agents are usually intelligent agents. An agent can be complex or simple. Agents interact within an environment. An agent should have certain properties. These properties are listed below [12]:

- *Autonomy*: An agent is autonomous if it tries to achieve goals on its own without direction from other agents. This means the agents have the ability to take their own decisions if unforeseen situations arise.

- *Knowledge Based*: This means agents take their decisions based on some knowledge. Agent should be able to justify their actions.

- *Rational*: An agents behavior is oriented on fulfilling its goals as completely as possible. So it tries to take decisions which are good for achieving its goal.

- *Social*: An agents is social if it purposefully does not hinder other agents in their tasks and actions or it supports other agents in achieving the overall goals of the system.

All the properties listed above are for a single agent. But the whole agent based system will not have all of these properties. Agent based systems have certain goals to achieve. Some general goals include higher efficiency, faster results, better safety and correctness, improved solutions, etc [12]. Some other goals include adaptability and reliability [12]. Agent based system are used in areas like artificial intelligence, distributed systems, robotics, gaming, psychology etc [12]. There are several platforms on which to implement multi-agent systems like Cougaar [15], JACK [16], 3APL [17]. This chapter will give details about three such systems JADE [14] , Cougaar [15] and CASA [1].

2.1.1 JADE

Java Agent Development Environment (JADE) is a software framework which implements multi agent systems. JADE is actually a middle-ware for the development of these systems. JADE implements Foundation for Intelligent Physical Agents (FIPA) specifications [13] and supports scalability in agent systems from debugging to deployment. JADE is created by "Telecom Italia (formerly CSELT) together with University of Parma in July 1998" [14, p.1]. JADE is developed in Java and is distributed under the LPG license [18]. Several industrial applications use JADE.

The main standout features of JADE over all the other platforms are [14]:

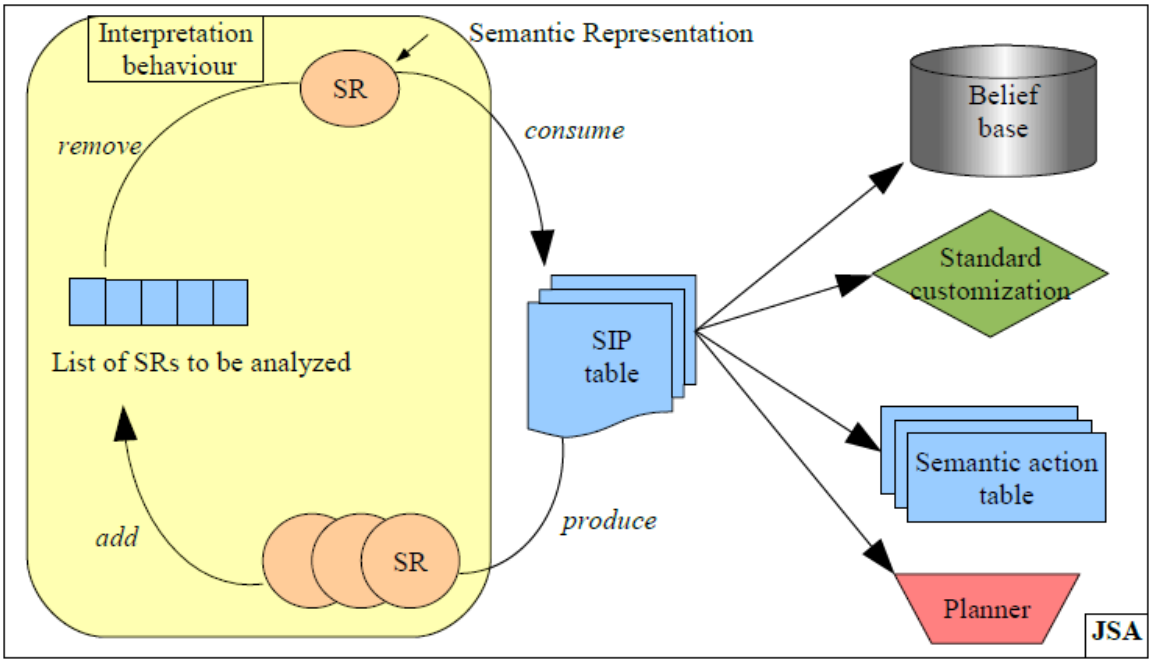

Figure 2.1: JADE Semantic framework [2, p. 4]

- JADE is completely based on FIPA specifications and follows FIPA specifications strictly.
- JADE provides set of functions that can be easily used while implementing multi-agent systems.
- JADE can be deployed on either of JEE; JSE or JME devices while others are not deployed on all of these devices.

2.1.1.1 JADE Semantic Extension

The JADE semantic extension is an add-on which adds the semantic capabilities of the FIPA-ACL language to the JADE platform. It is an implementation of FIPA-SL see (2.2.1.1). It checks syntactic and semantic consistency of messages [2]. It provides agents the capability to handle incoming messages and react to them according to FIPA standards automatically.

A JADE semantic agent is an agent built upon the Jade Semantic framework. Figure 2.1 illustrates the semantic framework and its components.

The framework consists of seven main components. These components are explained below [2]:

- *Semantic Representations* are FIPA-SL expressions representing the message. These are the same expressions as explained in FIPA-SL [19].
- The *belief base* stores the beliefs of an agent. A belief could be a predicate or a complex FIPA-SL formula.
- The *semantic action table* stores all the code for all the actions that agent can deal with. Each action has a semantic behaviour which performs the action.
- The *standard customization object* provides the programmer the ability to customize behavious for actions.
- The *planner* matches a plan to a goal.
- The *Semantic Interpretation Principles table* has all the FIPA principles for interpreting the messages. These principles handle semantic representations, produce new semantic representations, update the belief base etc.
- The *interpretation behaviour* is an algorithm which produces and consumes the semantic representations.

The belief base is the most important component for this research. The belief base stores facts believed by the agent according to the specific application domain. This component can only store and retrieve raw facts without actually understanding their meaning [2]. For example if an agent is informed that `(temperature 10)` is true, then he will correctly answer about `(any ?T (temperature ?T))`. But if the agent is queried about `(temperature_gt 15)`, he will not be able to answer as he does not know the semantic relationship between `temperature` and `temperature_gt` predicates [2].

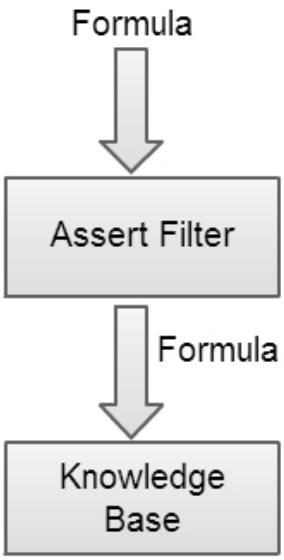

Figure 2.2: Assert Filter

The belief base has three referential operators as defined in Section 2.2.1.1 of FIPA-SL. Another operator is described below:

> "`some`, meaning find some objects o satisfying p, returns a list including all the objects o that can be found such as (B jsa (p o)) is true. Here B is the belief of jade semantic agent about p. If no such object can be found, it returns an empty list, but never returns null" [2, p.21].

The belief base provides a simple filter based mechanism to manage the beliefs. *Filters* are used to manage the access to the belief base. JADE semantic extension has two kinds of filters:

Assert filters, which are called when asserting a fact to the belief base. The main purpose of assert filter is to maintain the consistency of the belief base. These filters could be triggered before or after the assertion in the belief base. The formula to be asserted is passed to the assert filter and it returns a formula, which is the new formula to assert. For example, if the formula is a conjunction formula then the filter returns two separate formulas to be asserted

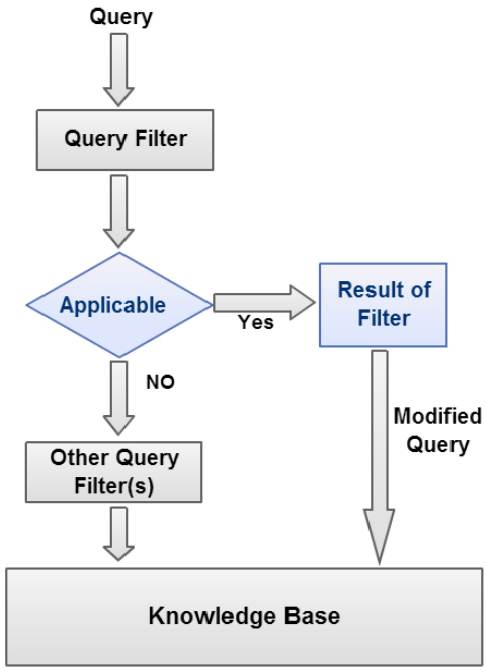

Figure 2.3: Query Filter

in the belief base. Figure 2.2 illustrates the working of an assert filter.

Query filters, are called when querying a fact or expression from the belief base. The output is either a list of match results or null. All matching facts found in the belief base, are stored in a list to be returned as output. If no matching fact is found in the belief base, null is returned. Figure 2.3 illustrates the working of a query filter.

This research will add the query filter called *ontology filter*. This filter will use the ontology on top of the knowledge from the belief base to produce the results for queries.

2.1.2 COUGAAR

Cougaar is a distributed agent based system having lot of features which provide scalability and reliability [15]. Cougaar is easy to use and reliable for complex applications as it has gone through rigorous stress and scalability testing [15].

2.1.2.1 Background

Cougaar is the output of an eight year US *Defence Advanced Research Projects Agency* (DARPA) funded effort to explore the potential of distributed multi-agent systems for military logistics. DARPA challenged a collection of companies and universities to build a detailed US military logistics plan for a major 180 day deployment [15].

Cougaar is good for large scale applications, which are data intensive. A lot of effort is done to maintain efficiency and security of the cougaar system. Cougaar applications are mostly distributed. Cougaar assumes the hardware to be unreliable and may fail. So, the system should be able to maintain itself on its own in case of failure. Cougaar also provides security in all forms like integrity, confidentiality and authenticity [15]. For achieving these goals, cougaar agents had multiple components. Agents are complex, heterogeneous and autonomous.

2.1.2.2 Architecture

Cougaar has been developed to follow certain principles. These principles are available as an open source software under a BSD - equivalent license [3]. Cougaar is a multi layered system. These layers include several logistics applications, and a collection of support infrastructure applications and a military grade security sub system [15]. All of these layers share the same underlying resources.

Cougaar is flexible as new components can be dynamically loaded into the system. Cougaar uses a multi-tiered interaction model. Agents interact via a local publish-and-subscribe mechanism and message passing is used for inter agent communication. Agents organize (form communities) to control their behaviours. Figure 2.4 shows the different components of the Cougaar system.

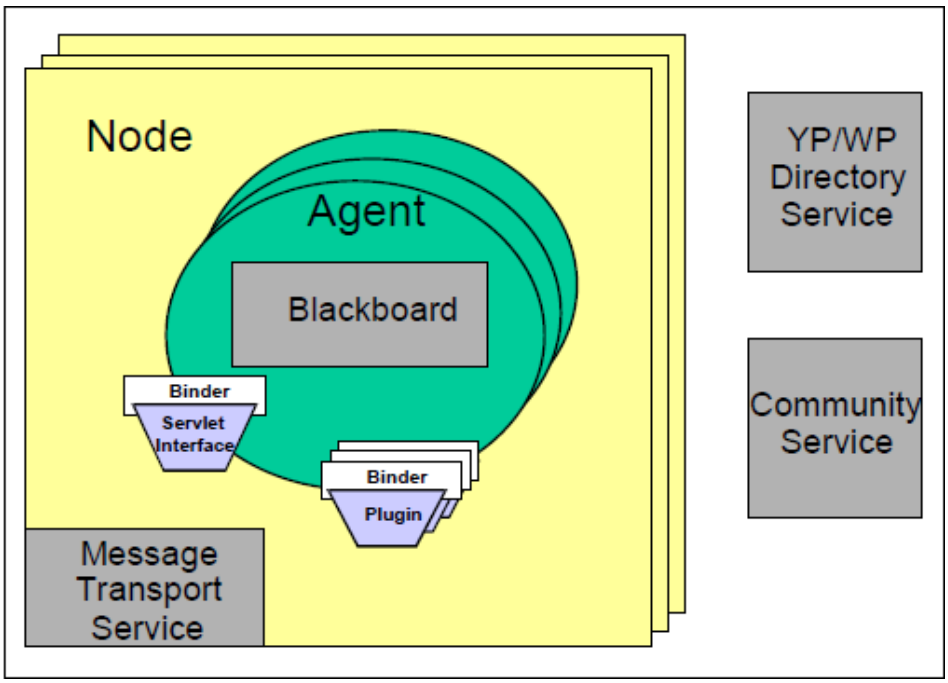

Figure 2.4: Cougaar Architecture [3]

Components are software units which interact with one another through abstract interfaces called *Services*. A *binder* is a wrapper around a service which hides all inner details about communication and the service itself. A *plugin* is another software component which is added based on the application requirements and specifications. A plugin is domain specific. An agent could have several plugins and overall behaviour is guided by all these plugins.

Message Transport Service (MTS) handles inter-agent communication. The mechanism used by MTS is quite robust and guarantees in-order delivery of messages in case of communication failures. Different components of agents interact via a *Blackboard* with standard publish/subscribe semantics. The whole data set of the system is the union of all blackboards [15].

Cougaar agents save their state by an mechanism called *Persistence*. All blackboard related and communication related information is saved as persistent data. If any failure occurs agents can reload using their persisted data. Agents with common goals or functionality form a group called a community.

Cougaar also has a *White Pages* service. This service stores a table having a map of agent names and network addresses. Cougaar also provides a *Yellow Pages* service. It is a directory service. Agents register their services in the directory and other agents can discover these services. A Service discovery is another component which is a best match provider across several yellow pages registries. Cougaar also has a *Logical Domain Model* for development of ontologies along with associated logic.

2.1.3 CASA

Collaborative Agent Systems Architecture (CASA) is a platform for developing agent based systems [1]. CASA can use different agent models for conversations among its agents and can support FIPA standards. CASA is written in Java 6.0 and has a Knowledge Base; which it has acquired from JADE. Programmers can define the behaviors of the agents using Java or Lisp code. Messages are interpreted in terms of *policies*. Policies are rules defined in Java or Lisp which execute a behavior [1].

2.1.3.1 Agent Structure in CASA

An agent in CASA has two threads. One of the threads listens for messages, decoding them in objects and queues them in the agent's event queue [1]. The second thread is the agent's main thread, dequeuing events and processing them. Figure 2.5 shows CASA's agent structure.

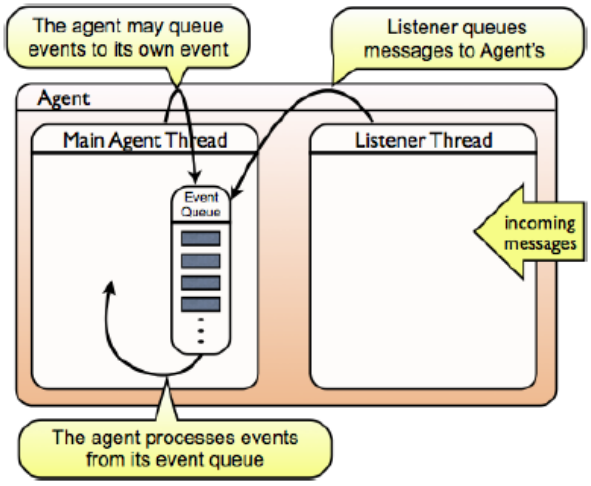

Figure 2.5: CASA agent structure [1]

Classes of agents are structured in a layered manner. The root is the `Thread` class. The most general agent class is `AbstractProcess` which inherits from the `Thread` class. The basic and concrete class called `TransientAgent` has knowledge about message semantics, ontologies, etc. The basic agent class structure is shown in Figure 2.6.

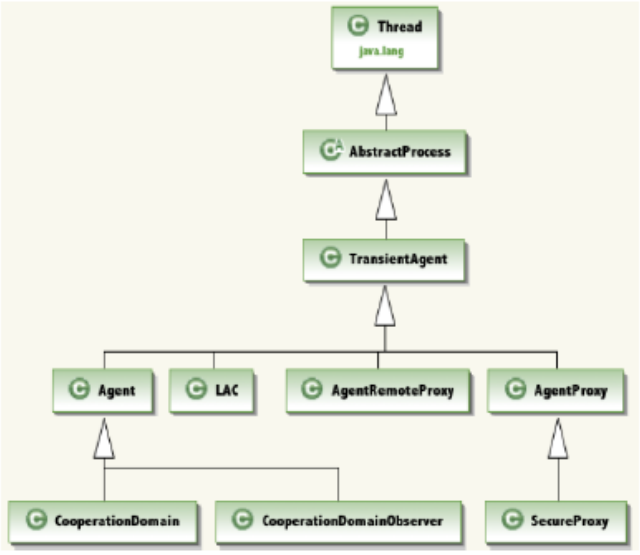

Figure 2.6: CASA agent class structure [1]

Each area has one *local area coordinator (LAC)*. The *LAC* is another class which inherits directly from `TransientAgent`. The LAC registers the agents, resolves their URLs and wakes

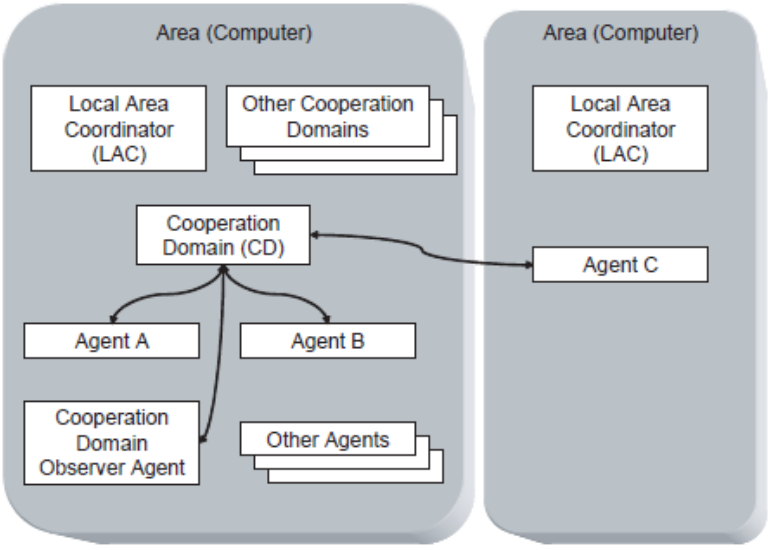

Figure 2.7: CASA LACs, CDs and normal Agents [1]

agents up as requested by a remote agent.

Cooperation Domains (CDs) are central point "hubs" for multi-agent conversations [1]. Message passing could be point-to-point, multi-cast or broadcast within the Cooperation Domain. Agents can also store persistent data in cooperation domains. Cooperation Domains can also store transaction histories for future use [1]. Figure 2.7 shows CASA's LACs and CDs.

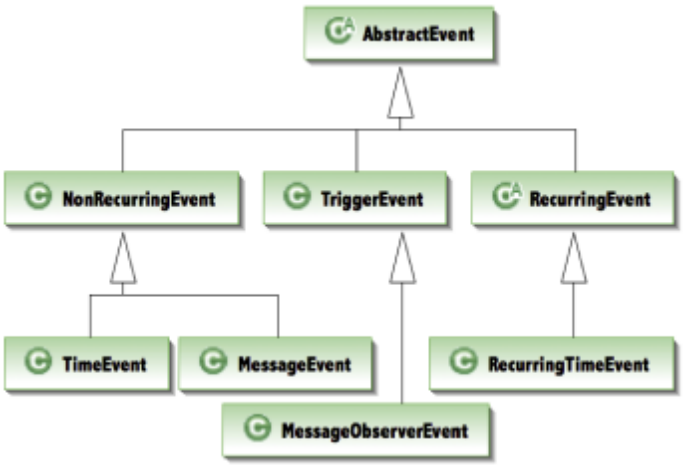

Figure 2.8: CASA event class [1]

2.1.3.2 Agent lifetime

An agent in CASA has a lifetime, consisting of initialization, event loop and and exiting. The first thing an agent does is initialization. After being initialized it enters an event loop. In this loop the agent processes the events from the event queue. If there are no events or commitments the agent goes to sleep. An agent exits when it receives an exit signal. An agent spends most of its time in the event loop processing events. An event can be anything from the receiving, sending, or observation of a message or changes in physical sensors etc [1]. Some of CASA's event classes are shown in Figure 2.8.

2.1.4 Comparison of JADE, Cougaar and CASA

Cougaar focuses on scalability and modularity. This is not the case with other multi-agent systems. In JADE and CASA each agent has its own thread. Cougaar also uses a thread pool with a powerful resource monitoring and allocation interface. Message passing in Cougaar is asynchronous. This makes it complex. JADE follows FIPA closely but cougaar does not follow FIPA strictly and CASA is scriptable to follow FIPA, variations of FIPA, or other protocols.

Another difference is the basic unit of development. In many multi-agent systems, an agent class is sub-classed to provide different behaviours. In Cougaar, one or more plugins are created which interact via a blackboard to provide different behaviours. Cougaar is a complex design compared to other multi-agent systems due to its modular approach. CASA is not so complex and easy to understand.

Overall, JADE is suited for simple applications that require FIPA compliance. Cougaar is suited for complex, customized and robust applications. CASA is intermediate between both of them. Also CASA is the system used in the Artificial Intelligence lab at University of Calgary. CASA provides all the components needed for this research, therefore it is used in this research work.

2.2 Logics

One needs some kind of logic to answer or reason about questions. Logic deals with rules for reasoning and arguing about things. The main aim of logic is to help distinguish between true and false [20].

Initially, logic dealt with arguments in natural languages used by humans [21]. For example: `All men are mortal`. But natural language turns out to be ambiguous. This means the same thing has different meanings in different sentences. Logic is sometimes described as the "calculus of computer science" [21]. Today computer science and logic are related in a lot of ways. Logic is used across a wide spectrum of areas of computer science, from artificial intelligence to software engineering. It provides powerful modeling and reasoning tools for computations.

"Logic is mainly concerned with two concepts: truth and provability" [20, p.1]. First order logic uses mathematical tools to prove the properties of a system. Logical systems have a language to write statements called *propositions* or *formulas* [20]. These formulas have well defined meaning or semantics. In logic, this meaning is defined by a truth value. So a formula could be either true or false [20]. This is termed propositional logic. Propositional logic is not powerful enough to represent everything in computer science and mathematics [20]. For example: `Not all birds fly` and `Some birds don't fly` are logically equivalent but propositional logic cannot capture their equivalence. So a more powerful logic is required to deal with this problem. Predicate logic is one such logic. Predicate logic is detailed in Section 2.2.1.

2.2.1 Predicate Logic

"A predicate is a verb phrase template that describes a property of objects, or a relationship among objects represented by the variables" [22, §Predicate]. For example, `The sky is blue` has `is blue` as a predicate and describes the property of being blue. Predicates have

a name. If `is blue` is given a name `B`, and we assert `B(x)`, this means `x is blue`.

Predicates with variables can be converted to propositions using a quantifier. There are two types of quantifiers: universal quantifier and existential quantifier [22].

The *Universal Quantifier* $\forall$ x Q(x) denotes the universal quantification of the formula Q(x). This means "For all x, Q(x) holds". $\forall$ is called a universal quantifier.

The *Existential Quantifier* $\exists$ x Q(x) denotes the existential quantification of the formula Q(x). This means "There exists an x such that Q(x) holds". $\exists$ is called an existential quantifier.

FIPA-SL is a modal predicate logic and is described in the subsequent section.

2.2.1.1 FIPA-SL

FIPA-SL stands for Foundation for Intelligent Physical Agents Semantic Language [19]. It has syntax and associated semantics to be used in conjunction with FIPA Agent Communication Language. FIPA-SL has content expressions to be used as the content of a message [19]. An expression could be a proposition, an action or an IRE [19].

> "A proposition, which may be assigned a truth value in a given context. Precisely, it is a well-formed formula (Wff) using the rules described in the Wff production.
>
> "An action, which can be performed. An action may be a single action or a composite action built using the sequencing and alternative operators. An action is used as a content expression when the act is request and other CAs derived from it.
>
> "An identifying reference expression (IRE), which identifies an object in the domain. This is the Referential operator and is used in the inform-ref macro act and other CAs derived from it." [19, p.5]

A *Well-formed formula* is formed from an atomic formula, whose meaning is determined by the semantics of the underlying domain. Relational operators in FIPA-SL are listed below:

- `(iota <term> <formula>)` "The iota operator introduces a scope for the given expression (which denotes a term), in which the given identifier, which would otherwise be free, is defined. An expression containing a free variable is not a well-formed FIPA-SL expression. The expression (iota x (P x)) may be read as the x such that P [is true] of x. The iota operator is a constructor for terms which denote objects in the domain of discourse." [19, p.7]

- `(any <term> <formula>)` "The any operator is used to denote any object that satisfies the proposition represented by formula." [19, p.9]

- `(all <term> <formula>)` "The all operator is used to denote the set of all objects that satisfy the proposition represented by formula." [19, p.10]

Another relational operator called `some` is explanined in section JADE semantic extension (2.1.1.1). The JADE semantic extension is an implementation of FIPA-SL. For more details about FIPA-SL refer to [19].

2.2.2 Description Logic

"Description Logics are a family of knowledge representation languages used for ontological modeling" [21, p. 1]. Description logics enable one to infer knowledge from facts stored in a knowledge base. This is termed as *reasoning*. The complexity of a description logic algorithm depends on the types of reasoning it can provide. So a lot of types of description logic are available. The choice of the description logic is largely dependent on the application at hand.

> "Description logics (DLs) provide means to model the relationships between entities in a domain of interest. In DLs there are

three kinds of entities: concepts, roles and individual names. Concepts represent sets of individuals, roles represent binary relations between the individuals, and individual names represent single individuals in the domain" [21, p. 2].

Description Logics have sets of statements called axioms which are true under a given situation. The axioms are separated into three groups: assertional (ABox) axioms, terminological (TBox) axioms and relational (RBox) axioms. ABox has concept assertions i.e, named individuals and their relation to concepts. Example: `Male(Tom)`, which asserts that individual Tom is an instance of concept Male. RBox contains role assertions i.e, relations between named individuals. Example: `brotherOf(Tom,Jack)`, which asserts that the individual named Tom is in the relation that is represented by brotherOf to the individual named Jack. TBox contains axioms describing relations between concepts. Example: `mammal` $\sqsubseteq$ `animal`, which means that the concept mammal is subsumed by the concept animal.

OWL2 is an implementation of several description logics and the relation between both is detailed in Section 2.3.2.

2.3 Ontologies

Communication is the most essential and integral part of our world. Good communication is important for people, industries, software systems, and even for software agents. People communicate in different languages to achieve their goals and tasks. Even if people communicate in the same language, they refer to the same thing in different ways. For example one person uses the word *search* and other uses the word *explore*. Both of them belong to the same language but are synonyms of each other. This is the case in human communication, but this is not as intuitive in the case of software systems like agent based systems. For having good communication among software systems, the system developers should have the same understanding of concepts. This means systems should have well defined concepts

and terms. This task is challenging as software systems vary a lot in viewpoints, work flow, methodology, assumptions, etc. Software systems could vary on interpretation of the concepts and concepts could mismatch very easily. This could lead to misunderstanding of requirements and specifications of the system [11] as concepts vary in their meaning. So, the concepts should be well defined to avoid this misunderstanding in software systems.

Systems require shared understanding of concepts. This could be achieved using ontologies.

According to Mike Usehold and Michael Gruninger:

> "Ontology is the term used to refer to the shared understanding of some domain of interest which may be used as a unifying framework " [11, p.5].

An ontology may take different forms. It could include terms and their meanings (i.e definitions) [11]. "In philosophy, ontology is a study of the kind of things that exist" [11]. Ontology is always based on two things. First the domain or the subject we are working with. Secondly knowledge about that domain [11]. Different researchers in *Artificial Intelligence* have tried to explain ontology as a content theory. McCarthy and Hayes' theory (epistemic versus heuristic distinction) [23], Newell's theory (Knowledge Level versus Symbol Level) [24] try to define ontology as a content theory in their own ways. Ontology can be represented as a type lattice, but often is extended to include non-subsumption relationships.

2.3.1 Ontology engines

Ontology engines support the design and specification of ontologies. Some of the engines are Ontolingua and OWL 2 and these are described in the subsequent sections.

2.3.1.1 Ontolingua

> "Ontolingua provides a distributed collaborative environment to browse, create, edit, modify, and use ontologies. Ontolingua

> supports over 150 active users, some of whom have provided us with descriptions of their project" [25, §Software Description].

Ontolingua was an extension of knowledge representations techniques [26]. Ontolingua was designed by Gruber [26].

> "Gruber extended KIF with additional syntax to capture intuitive bundling of axioms into definitional forms with ontological significance; and a Frame Ontology to define object-oriented and frame-language terms. The Ontolingua Server has extended the original language in two ways. First, it provides explicit support for building ontological modules that can be assembled, extended, and refined in a new ontology. Second, it makes an explicit separation between an ontologys presentation and representation" [26, p.4].

For more details about ontolingua refer to [26].

2.3.1.2 OWL2

The World Wide Web is continuously growing and some tools are required to maintain this huge pool of data [27]. Information should be described in a way easily understandable by both machines and humans. OWL is a set of knowledge representation languages [27]. OWL has three sub-languages. These are listed below [27]:

- OWL Lite, used for classification hierarchies and simple constraint features.
- OWL DL used for providing maximum expressiveness in reasoning systems. DL here stands for Description Logics (§2.2.2). Description Logics is a set of knowledge representation languages. It is used for formal reasoning of concepts in the ontology. So OWL DL has its roots in description logics also.

- OWL Full provides maximum expressiveness and the syntactic freedom of Resource description framework (RDF). RDF is a data interchange model for applications on the web. RDF is useful for linking data which differ in the underlying schema.

These languages have the following relations [27, p.5]:

- "Every legal OWL Lite ontology is a legal OWL DL ontology.
- "Every legal OWL DL ontology is a legal OWL Full ontology.
- "Every valid OWL Lite conclusion is a valid OWL DL conclusion.
- "Every valid OWL DL conclusion is a valid OWL Full conclusion."

OWL is an extension of RDF. OWL has been taken over by OWL2. It is also an ontology language for the semantic web. OWL2 ontologies can also be used with RDF. Figure 2.9 gives an overview of the building blocks of OWL2. Figure 2.9 also shows relations of these blocks to each other. The primary exchange syntax for OWL 2 is RDF/XML [4]. "While RDF/XML provides for interoperability among OWL 2 tools, other concrete syntaxes may also be used. These include alternative RDF serializations, such as Turtle, an XML serialization, and a more "readable" syntax, called the Manchester Syntax, that is used in several ontology editing tools. Finally, the functional-style syntax can also be used for serialization, although its main purpose is specifying the structure of the language" [4, p.3].

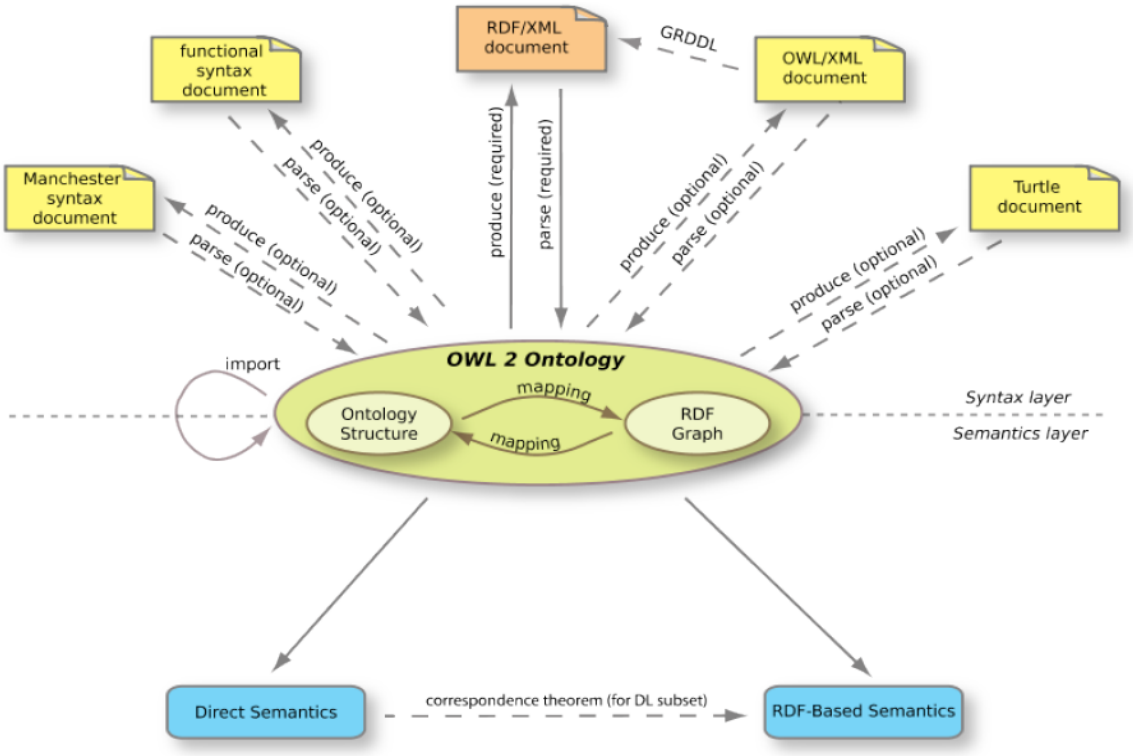

Figure 2.9: Structure of OWL2 [4, p. 3]

OWL2 ontologies consist of the following three different syntactic categories [5]:

- Entities, like classes, properties and individuals. These are identified by Internationalized Resource Identifiers (IRIs). An IRI is a string representing a unique identifier for an entity in the ontology. In OWL, Uniform Resource Identifiers (URIs) were used for this same purpose. A class is like a type in the ontology and a property represents a relation of the type. For example, a class dog can be used to represent the set of all dogs. Similarly, the object property parentOf can be used to represent the parent-child relationship. Finally, the individual Rover can be used to represent a particular dog called "Rover".

- Expressions represent complex notions in a particular domain of interest. This could include restrictions on individuals.

  For Example, `Declaration(Class(biology:animal))`, is an expression spec-

ifying that animal is a class in the biology ontology.

- Axioms are true statements in the domain. For example, the subclass axiom can be used to state that cat is a subclass of the animal class.

OWL2 adds new functionality to OWL, like [5]:

- keys
- property chains
- richer data types
- qualified cardinality restrictions
- asymmetric, reflexive and disjoint properties
- enhanced annotation capabilities.

OWL2 ontologies also support modularization. This means an OWL2 ontology O can import another OWL2 ontology O1 and get access to all the entities, expressions and axioms of O1. An example of an import is:

```
Ontology(<http://casa.cpsc.ucalgary.ca/casa.TransientAgent.owl>
Import(<http://casa.cpsc.ucalgary.ca/ontologies/casa.owl>)
```

OWl2 has prefix names associated with their corresponding IRI's. Table 2.1 shows the Standard prefix names in OWL2:

Table 2.1: OWL2 standard Prefix Names

Prefix name	Prefix IRI
rdf:	⟨http://www.w3.org/1999/02/22-rdf-syntax-ns#⟩
rdfs:	⟨http://www.w3.org/2000/01/rdf-schema#⟩
xsd:	⟨http://www.w3.org/2001/XMLSchema#⟩
owl:	⟨http://www.w3.org/2002/07/owl#⟩

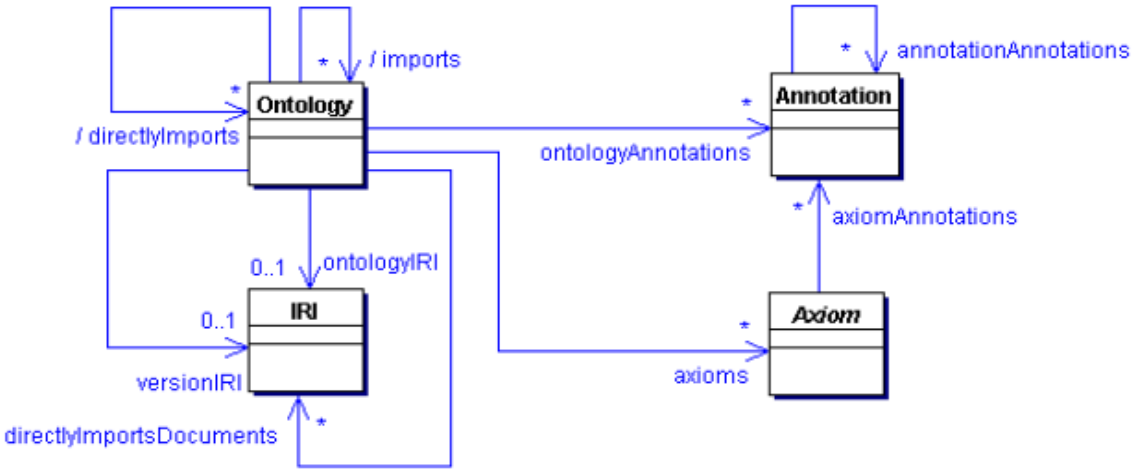

Figure 2.10: OWL2 ontology structure [5, p. 8]

An OWL2 ontology follows structural specification of OWL2 shown in Figure 2.10. An ontology is a set of axioms. Ontology also contains annotations and can import other ontologies.

An ontology IRI is an unique identifier for an ontology. An ontology may also have a version IRI. This version IRI may or may not be equal to the ontology IRI. OWL 2 is the ontology engine used in this research.

2.3.2 OWL relationship to Description Logic

OWL is one of the applications of description logic. The main components of OWL are similar to those of description logic. In OWL concepts are called classes and roles are called properties. Description logic has had a major influence on the development of OWL and its features. OWL's syntactic component is called OWL DL. A large part of OWL DL is a variant of description logic. For example: The description logic axiom `Mother` $\equiv$ `Female` $\sqcap$ `Parent` in OWL is represented as `EquivalentClasses(Mother ObjectIntersectionOf(Female Parent))` [21]. This example illustrates the close relationship between OWL and description logic. So,

when converting from description logic to OWL (Functional-Style Syntax), operators are translated to the corresponding operator name (written as a function name). OWL has more operators as compared to description logic. OWL also provides expressive features like support for data types and data type literals. OWL also has some other features which are not in description logic. OWL provides means to name an ontology and import axioms form other ontologies. OWL also includes comments called *annotations*.

2.3.3 Uses of ontologies

Ontologies provide the subsumption relation between the terms. This relation is useful for this research work. Example: The term `animal` subsumes the term `dog`. That means `all dogs are animals`. Ontologies also enable shared understanding of various concepts of a software system. This enables people to build systems which are efficient and have better communication structure. So, using ontologies improves software systems. Ontologies create a set of relationships among software systems. Systems using knowledge from multiple domains could really benefit from this use of ontologies. Distributed agent based systems also benefit from this use of ontologies. Ontologies could be reused among several software systems. This saves a lot of time and effort to redo things [11].

2.3.4 Building ontologies

Certain things should be kept in mind when building an ontology. Some informal concepts are discussed here. The following points should be kept in mind when we are building ontologies [11]:

- Terms, relationships and types should be clearly defined. These should not be ambiguous. So a lot of time and effort should be put in this step. The more time put in this step helps to easily maintain the ontology.

- The definitions given to terms or axioms should be consistent. They should not vary from one scenario to another. This is important as ontologies are to be reused across several systems. Inconsistency hampers use of ontologies across different systems.

Ontologies should be easy to extend and easily upgradable, meaning new terms, types or relationships are easy to add. Steps for building ontology are listed below [11]:

1. *Finding Concepts*: This means to determine the different concepts in the domain of software application. These could initially be really general in nature. Brainstorming and grouping with domain experts are some of the techniques used to determine these underlying concepts.

2. *Define Concepts*: After having an initial list of concepts, we then define these concepts. Definitions should as clear as possible. This will lead to better performance of the system. These definitions should be consistent across different systems.

3. *Building a Meta Ontology*: Carefully determine the relationships between different concepts. Define all of these in terms of a subsumption tree. This meta ontology could be defined using *natural language definitions*.

4. *Code or formally build Ontology*: The output of previous step is then coded to form an ontology. Different formal methods could be used to code the ontology. These vary from system to system.

2.3.5 Projects using ontologies

This section will discuss certain projects which are use ontologies. The projects discussed here are from different application domains.

2.3.5.1 STEP

STEP [11] stands for Standard for the exchange of Product Model Data. "It is an inter lingua to define and specify products" [11, p. 42]. STEP is used for achieving interoperability and also enables the data to be exchanged among different computer systems. It also helps an organization to store data in a common computer interpretable format. STEP uses the formal specification language EXPRESS to give information about a product.

> "EXPRESS is a standard data modeling language for product data. It can be used to express information or knowledge or systems in a structure that is defined by a set of rules. It provides both textual and graphical representations for the concepts" [29, §Overview]

The overall goal of STEP is to create a database of the product information. STEP uses XML schema model for storing the information about the products. The XML schema is the most primitive type of ontological model. The XML model was used because a lot of business at that time were using XML modeling. STEP uses a library of XML specifications including Document Type Definitions and XML Schema. This library helps to categorize the products by giving them a name, description and unique identification.

2.3.5.2 CORBA

CORBA [11] stands for Common Object Request Broker Architecture. "CORBA is used to retrieve objects and invoking operations on objects across a network" [11, p.37]. This project is an collaborative effort done by Object Management Group. CORBA is based on a mechanism by which objects make requests and get responses for those requests. The Object request Broker is a module which provides interoperability between applications running in heterogeneous environments. The Object request broker is supported by a Interface Definition Language (IDL) module. The IDL specifies objects and operations to the Object request Broker. IDL is supported by the KSL Ontology server. The KSL server provides a

translator between IDL and Ontolingua. CORBA also contains informal notions of ontology. In this project Business Object Management Group has made a glossary of terms to be used in object Model. This basically provides a shared understanding of objects of the system.

2.3.6 Implemented project ontologies

This section will discuss certain projects addressing the problem of implementing ontologies.

2.3.6.1 CYC

CYC [11] is a project by Microelectronics and Computer Technology Corporation (MCC). In this project they are building ontologies for a different type of application domain. CYC has a knowledge base which stores knowledge in the form of assertions. The knowledge base language used is CycL, which is based on predicate calculus and has a syntax similar to that of the Lisp programming language [30]. An example predicate for the knowledge base is given below:

`(#$isa #$Mike #$HockeyPlayer)`, meaning Mike belongs to the collection of HockeyPlayers.

CYC also uses ontologies. Ontologies are termed as microtheories [11]. A microtheory could capture knowledge about a domain. A single domain could have multiple microtheories. So, it is like a network of microtheories across a set of domains. Thus it provides knowledge across different domains using microtheories. The limitations of this work include its complexity and little documentation.

2.3.6.2 TOVE

TOVE [11] stands for Toronto Virtual Enterprise. This project implements an enterprise ontology. The main objective is to use a common and shared terminology for various enterprises. The terms defined in the ontology should have unambiguous meaning. Semantics for ontologies are implemented using Prolog axioms. TOVE also has a symbology i.e. symbols for terms and concepts in the system. The project provides classes of enterprises based on

the assumptions of the system. Overall the project aims at providing a enterprise ontology model.

2.3.6.3 KACTUS

KACTUS [11] is a European ESPRIT project. It supports the reuse of knowledge across different systems. It provides a methodology for this by using a knowledge base. This knowledge base is used for designing, operations, maintenance and redesigning of the systems. KACTUS creates domain ontologies and reuses them across different applications. KACTUS is also able to integrate with existing standards like STEP. KACTUS uses Conceptual modeling language (CML). CML is used here to define the difference between domain knowledge, inference knowledge and task knowledge. KACTUS has a user interactive interface for browsing and editing ontologies. The KACTUS toolkit has several packages for theoretical and practical issues. KACTUS also provides support for EXPRESS [§2.3.5.1] and Ontolingua [§2.3.1.1].

2.3.6.4 Plinus

"The Plinus project aims at semi automatic knowledge extraction from short natural language text" [11, p.53]. The inputs for Plinus are title and abstract fields of bibliographic document descriptions. These inputs are taken from the web. The core of the Plinus project is the use of ontologies. Plinus uses a set of ontologies to cover concepts from multiple domains. Some parts of the ontology have been designed in a top-down fashion. Other parts of the ontology have been developed in a bottom-up fashion. The domain knowledge here is expressed as concepts. Lexicons are used to express the semantics of the language. The lexicons map concepts from natural language to formal expressions in the knowledge representation language. The ontology in the project also specifies the output of the language dependent process. This means an input not having a concept in an ontology, cannot appear in the output.

2.4 Knowledge Bases

A KB is useful for knowledge management [8]. A KB could be either machine-readable or could be used by humans. In agent based systems machine readable KB's are used. Computer systems can apply deductive rules on this KB to derive new knowledge. So, a knowledge base is an intelligence tool for producing intelligent decisions [9]. Systems using KB's for accessing and acquiring knowledge are termed as knowledge-based systems. Each agent-base system has a knowledge base language. The knowledge base language used by agent based systems like CASA and JADE is FIPA-SL [19]. FIPA-SL is a content knowledge base language [19]. Some detail about the FIPA-SL are listed in Section (2.2.1.1). Most agent-based system are knowledge-based systems. Figure 2.11 shows a typical knowledg-based system architecture.

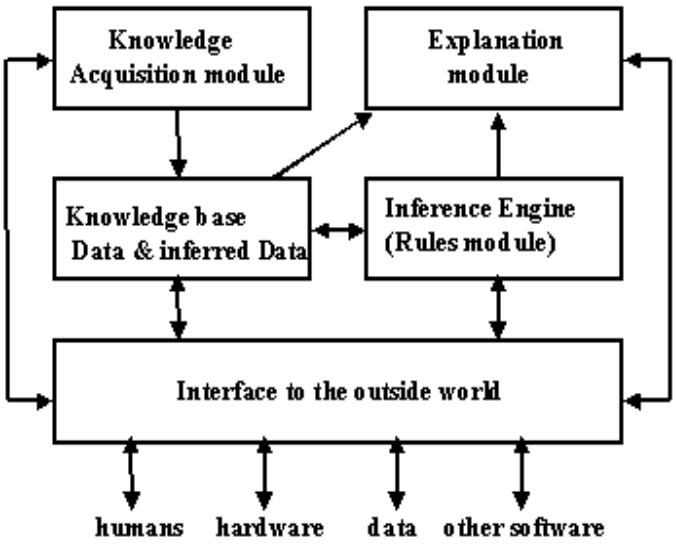

Figure 2.11: Knowledge-based system [6, §Knowledge Based Systems]

From Figure 2.11, it can be seen that most of the parts actually refer to the knowledge base for information. So, the knowledge base is the core of such systems. The inference engine uses the information from the KB to infer knowledge and take intelligent decisions.

Since other parts use the information from the KB, its important to think about how to store information in a KB. This task is called knowledge representation. A knowledge base

should store *facts*. Facts are truths about the real world. Facts are represented in terms of *symbols*. These symbols could be manipulated by the systems. Natural language is one way to represent the facts. For example:

`kitty is a cat`

`all cats have tails`

Other ways to represent knowledge include logics, semantic networks and production rules and these are described in the subsequent sections.

2.4.1 Predicate Logic

Predicate logic uses a formal language to represent and mathematically manipulate knowledge. Mathematical deductions are used to derive new knowledge from old knowledge. Some operators in predicate logic are:

- The logical *and* which is termed as the *conjunction* of two logical propositions.
- The logical *or* which is termed as the *disjunction* of two logical propositions.
- The negation, which is the *not* unary operator.

The previous natural language example can now be represented as:

```
cat(kitty)
```

`all cats have tails` could be represented as:

```
cat(x) -> hasatail(x)
```

Then following could be deduced:

```
hasatail(kitty)
```

This means

`kitty has a tail`. For more detail about predicate logics see Section 2.2.1.

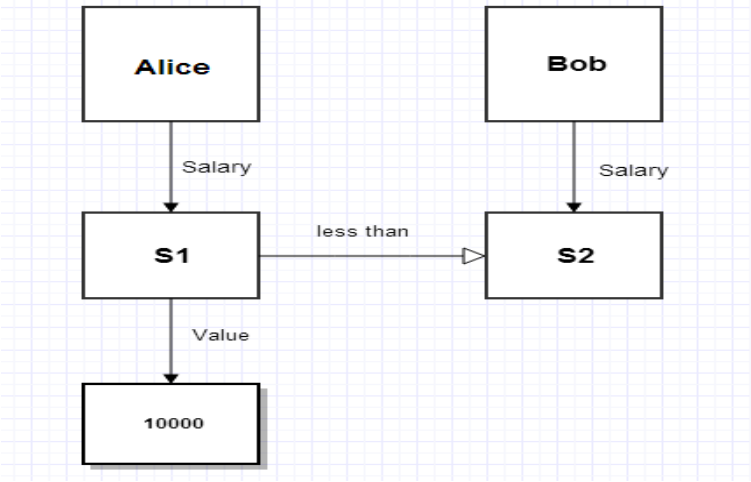

Figure 2.12: Semantic Net

2.4.2 Semantic Net

The *Semantic Net* is a graphic language for representation of facts [9]. Facts represent entities which could be reasoned about. Figure 2.12 shows a semantic net. In this example, Alice has a salary S1 and Bob has a salary S2. Also it is represented that S1 is less than S2. The value of S1 is given to be 10000. So it can be deduced that Bob's salary would be greater than 10000.

2.4.3 Production Rules

Production Rules are based on the idea of *condition-action* pairs. Condition-action pairs are also called *if-then* pairs or just *productions*. Production rules are written as "if this condition holds, then this action is appropriate". Here a condition could be a boolean evaluation and the resultant consequent could be an action or a state. They are powerful form of knowledge representation. Consider the example:

`if the temperature is above 30 degree Celsius then conclude it is sunny`.

Production Rules are evaluated either by *forward chaining* or *backward chaining*. In *forward chaining*, reasoning starts from the facts already present in the KB, working forward through the inference engine and discovering conclusions from the data. In *backward chain-*

ing, a decision is picked and worked on in a backward direction to justify it by the facts in the KB.

2.5 Summary

This chapter focused on various aspects of this research. Agent based systems were discussed. Three such systems were discussed JADE [14], Cougaar [15] and CASA [1]. These systems were then compared and reasons for using the CASA system in this research was presented. Logic was discussed, including description and predicate logic. Ontologies, knowledge bases, and knowledge base languages (FIPA-SL) was also discussed.

Chapter 3

Requirements

3.1 Introduction

The purpose of this chapter is to present the requirements for this research work. The requirements include system requirements and testing requirements. All of these requirements are discussed in detail in the coming sections.

3.2 System requirements

R.1 A knowledge based system

First of all, in this research, a system that uses a knowledge base is required. A knowledge base should store facts about the real world. Section 2.4 has details about knowledge bases.

R.2 System using ontologies

Another requirement is a system that uses ontologies. Ontologies are explained in detail in section 2.3. Ontologies express the relational knowledge of a domain. This means ontologies include *types, individuals and relations* in a domain. Ontological knowledge could be used on top of the knowledge from a knowledge base to provide better query results.

R.3 A knowledge base editor

The system should have a way to create and display the knowledge in a knowledge base. The system should have a way to assert the knowledge in the knowledge base.

R.4 An agent-based system

Another requirement is to choose an agent-based system which incorporate all of the previous requirements. An agent should have a knowledge base. The system should provide a way to display the current state of the knowledge base of each agent. This means it should have commands to display, add and modify the knowledge in the knowledge base of agents. Each agent should have an ontology associated with it. It should be able to modify the ontology. This means it should be able to add new types and relations to the ontology. Each agent should have an interface. The interface of an agent should be easy to use and navigate. It should help the user to easily navigate the features. It should display the current state of the knowledge base. Also, the interface should display the ontology. This means the interface should be capable of displaying the relations, types and individuals in the ontology.

3.3 Test Requirements

The ontology filter will be tested against a knowledge base and a variety of query expressions. The system will be tested with and without ontology filter. Before discussing the test requirements, let us discuss *reverse subsumption*. This term will be used in some test requirements. Normally in a type lattice, if a type A subsumes a type B then B specializes A i.e, every individual of type B is also of type A. But if we consider the predicates instead of types then the subsumption is replaced by reverse subsumption. In that case, instead of specialization, generalization is followed [31, p. 22]. For this research, the first identifier of a query expression (enclosed in parenthesis) serves as a predicate identifier. So, reverse subsumption is applied on that identifier and all the other identifiers have normal subsumption. Consider an ontology having types `animate`, `animal`, `mammal` and `corporeal`. `corporeal` subsumes `animate`, `animate` subsumes `animal` and `animal` subsumes `mammal`. The knowledge base has a fact `(animate animal)`, then the query `(animate mammal)` will

Figure 3.1: Type Hierarchy

result `true` and query `(animate corporeal)` will result `false`. If the knowledge base has a fact `(animate animal)`, then the query `(corporeal animal)` will result `true` and query `(mammal animal)` will result `false`. In these examples, the first identifier follows reverse subsumption and second identifier follows regular subsumption.

3.3.1 First Test requirement

R.5 Direct and indirect type subsumption in a zero-argument query

Consider an ontology having types P0, P1, P2 and P3 where P0 :>P1 :>P2 :>P3 (:>means subsumes). The type hierarchy is shown in Figure 3.1.

Suppose, the knowledge base has a fact `P1`. P1 is a type identifier. The system will be queried for `?P2` and using the ontology filter we should get direct type subsumption. This means the result for this query should be `true` as P1 subsumes P2. In this case normal subsumption takes place meaning the agent looks for the parents of the identifier in the knowledge base. On the other hand, if the knowledge base is queried about P0 like ?P0 then the result should be `nil` (means no knowledge) as no parent of P0 is in the knowledge base. Table 3.1 gives all the possible test cases (queries) and expected output. This table assumes that the knowledge base has a fact `P1`.

Consider, the knowledge base to have a fact `(P2)`. P2 is an predicate identifier. The system will be queried for `?(P1)` and using the ontology filter we should get direct type

Table 3.1: Test cases for subsumption of zero argument with the fact `P1` in the KB

Query Expression	With Filter	Without Filter
P0	nil	nil
P1	T	T
P2	T	nil
P3	T	nil

subsumption with zero arguments. This means the result for this query should be `true` as P1 subsumes P2. In this case reverse subsumption takes place meaning the agent looks for the children of the identifier in the knowledge base. On the other hand, if the knowledge base is queried about P3 like ?(P3) then the result should be `nil` as no child of P2 is in the knowledge base. Another case here would be, if the query is made about P0 like `?(P0)` then the result again should be `true` and this is called indirect type subsumption with zero arguments.

If no ontology filters were used then the only test case that returns true is `(P2)`, as it is present in the KB. All other test cases should return `nil`. Table 3.2 gives all the possible test cases (queries) and expected output. This table assumes that the knowledge base has a fact `(P2)`.

3.3.2 Second Test requirement

R.6 Direct and indirect type subsumption in a one-argument query

Consider an ontology having types T0, T1, T2 and T3 where T0 :>T1 :>T2 :>T3. The type hierarchy is shown in Figure 3.2. The ontology also has predicates shown in Figure 3.2.

Table 3.2: Test cases for subsumption of zero argument with the fact `(P2)` in the KB

Query Expression	With Filter	Without Filter
(P0)	T	nil
(P1)	T	nil
(P2)	T	T
(P3)	nil	nil

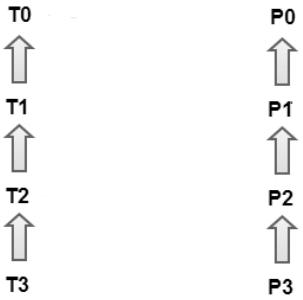

Figure 3.2: Type Hierarchy

Suppose the knowledge base has a fact `(P1 T1)`. The first argument of the query follows reverse subsumption as the previous test requirement and the second argument follows normal subsumption. The system will be queried for `?(P1 T2)` and using the ontology filter we should get direct type subsumption of the first argument. This means the result for this query should be `true` as T1 subsumes T2. This is normal subsumption meaning the parents of the identifier are searched in the knowledge base. On the other hand, if the knowledge base is queried about T0 then the result should be `nil`. Another case here would be if the knowledge base has a fact `(P1 T1)` and the query is made about T3 like `?(P1 T3)` then the result again should be `true` and this is called indirect type subsumption of the first argument.

If no ontology filters was used then the only test case which returns `true` is `(P1 T1)` (as it is present in the knowledge base). All other test cases returned `nil`. Table 3.3 gives all the possible test cases (queries) and expected output. This table assumes that the knowledge base has a fact `(P1 T1)`.

3.3.3 Third Test requirement

R.7 Direct and indirect type subsumption in a two-argument query

Consider an ontolog shown in Figure 3.2. Suppose the knowledge base has a fact `(P1 T1 T1)`. The first argument follows reverse subsumption and the other two arguments follow the

Table 3.3: Test cases for subsumption of one-argument with the fact `(P1 T1)` in the KB

Query Expression	With Filter	Without Filter
(P0 T0)	nil	nil
(P0 T1)	T	nil
(P0 T2)	T	nil
(P0 T3)	T	nil
(P1 T0)	nil	nil
(P1 T1)	T	T
(P1 T2)	T	nil
(P1 T3)	T	nil
(P2 T0)	nil	nil
(P2 T1)	nil	nil
(P2 T2)	nil	nil
(P2 T3)	nil	nil
(P3 T0)	nil	nil
(P3 T1)	nil	nil
(P3 T2)	nil	nil
(P3 T3)	nil	nil

normal subsumption. The system will be queried for `?(P1 T1 T2)` and using the ontology filter we should get the direct type subsumption of the second argument. This means the result for this query should be `true` as T1 subsumes T2. On the other hand, if knowledge base is queried about T0 then the result should be `nil`. Another case here would be if the knowledge base has a fact `(P1 T1 T1)` and the query is made about T3 like `?(P1 T1 T3)` then the result again should be `true` and this is called indirect type subsumption of the first argument.

If no ontology filters was used then the test case which returns `true` is `(P1 T1 T1)` (as it is present in the knowledge base). All other test cases returned `nil`. Table 3.4 gives all the possible test cases (queries) and expected output. This table assumes that the knowledge base has a fact `(P1 T1 T1)`.

3.3.4 Individuals Test requirement

R.8 Direct and indirect subsumption over individuals

Consider an ontology shown in Figure 3.3. I0, I1, I2 and I3 are individuals of classes T0,

Table 3.4: Test cases for subsumption of two-argument with the fact `(P1 T1 T1)` in the KB

Query Expression	With Filter	Without Filter
(P0 T1 T0)	nil	nil
(P0 T1 T1)	T	nil
(P0 T1 T2)	T	nil
(P0 T1 T3)	T	nil
(P0 T0 T0)	nil	nil
(P0 T0 T1)	nil	nil
(P0 T0 T2)	nil	nil
(P0 T0 T3)	nil	nil
(P0 T2 T0)	nil	nil
(P0 T2 T1)	T	nil
(P0 T2 T2)	T	nil
(P0 T2 T3)	T	nil
(P0 T3 T0)	nil	nil
(P0 T3 T1)	T	nil
(P0 T3 T2)	T	nil
(P0 T3 T3)	T	nil
(P1 T1 T0)	nil	nil
(P1 T1 T1)	T	T
(P1 T1 T2)	T	nil
(P1 T1 T3)	T	nil
.....	...	...
.....	...	...
.....	...	...
(P3 T3 T0)	nil	nil
(P3 T3 T1)	nil	nil
(P3 T3 T2)	nil	nil
(P3 T3 T3)	nil	nil

T1, T2 and T3 respectively.

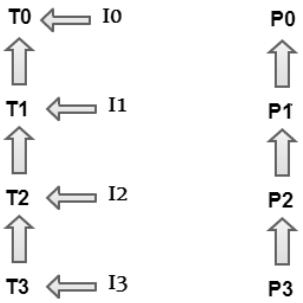

Figure 3.3: Type Hierarchy

Table 3.5 gives all the possible test cases (queries) and expected output. The knowledge base will have facts `T1`, `(T1)`, `(P1 T1)` and `(P1 T1 T1)`.

If the query is `?I1` then the result should be `true` as it is in the KB and if the query is `?I0` the result should be `nil` as it is not in the knowledge base. If the query is `?(P0 I1)` then the result should be `true` as a fact about T1 is in the knowledge base and if the query is `?(P0 I0)` the result should be `nil`. If the query is `?(P0 I1 I1)` then the result should be `true` as a fact about T1 is in the knowledge base and if the query is `?(P0 I1 I0)` the result should be `nil`.

3.3.5 Complex Test cases

The ontology filter is also tested with other filters like *and*, *or*, *not*. For performing tests, an agent's knowledge base is asserted with facts `(P1)`, `(P1 T1)` and `(P1 T1 T1)`. Table 3.6

Table 3.5: Test cases for subsumption over individuals with the facts `T1`, `(T1)`, `(P1 T1)` and `(P1 T1 T1)` in the KB

Query Expression	With Filter	Without Filter
I1	T	T
I0	nil	nil
(P0 I1)	T	nil
(P0 I0)	nil	nil
(P0 I1 I1)	T	nil
(P0 I1 I0)	nil	nil

Table 3.6: Complex test cases with the facts `(P1)`, `(P1 T1)` and `(P1 T1 T1)` in the KB

Query Expression	With Filter	Without Filter
(! (P1 T2))	nil	T
(! (P2 T0))	T	T
(and (P1 T2) (P1 T1))	T	nil
(and (P1 T2) (P0 T0))	nil	nil
(or (P1 T2) (P1 T1))	T	T
(or (P0 T0) (P2 T1))	nil	nil

gives all the possible test cases (queries) and expected output.

In the test cases listed in table 3.6, queries are made against *and*, *or*, *not* filters. In case of `not (!)` filter the result of the query is simply inverted. In the case of the `and` filter, if the two queries return `true` then the result is `true`. In case of the `or` filter, if the two queries return `false` then the result is `nil`.

3.4 Summary

This chapter has outlined the basic requirements to add the ontology filter to a knowledge based system. Table 3.7 gives a summary of all the requirements. The next chapter explains the implementation of these requirements.

Table 3.7: Requirements

Number	Detail
R.1	A knowledge based system
R.2	System using ontologies
R.3	An knowledge base editor
R.4	An agent-based system
R.5	Direct and indirect type subsumption in a zero-argument query
R.6	Direct and indirect type subsumption in a one-argument query
R.7	Direct and indirect type subsumption in a two-argument query
R.8	Direct and indirect subsumption over individuals
R.9	Complex test cases

Chapter 4

Design and Implementation

4.1 Introduction

This chapter describes the design and implementation of the ontology filter in the CASA agent based system. The description includes references back to the requirements in the previous chapter. Section 4.2 gives details about the design of the filter. Section 4.2.2 gives the algorithm for searching through the ontology. Section 4.2.3 gives the complexity analysis of the algorithm. Section 4.3 describes the implementation details of the research.

4.2 Design

The design for this research work meets the requirements listed in Chapter 3. First of all, an agent based system was chosen which could fulfill all the requirements related to this research work. CASA was chosen as it uses ontologies and a knowledge base. CASA fulfills both of those requirements. CASA uses the OWL2 ontology engine and the JADE semantic extension knowledge base. This knowledge base (belief base) is rigid as it could only give results for the facts it knows about. This was not useful. So CASA is a good fit for adding ontology filter to the knowledge base to make it more useful. Also, the agents in CASA have an interface which fulfills the interface requirements from Chapter 3. The next section describes the design of the filter.

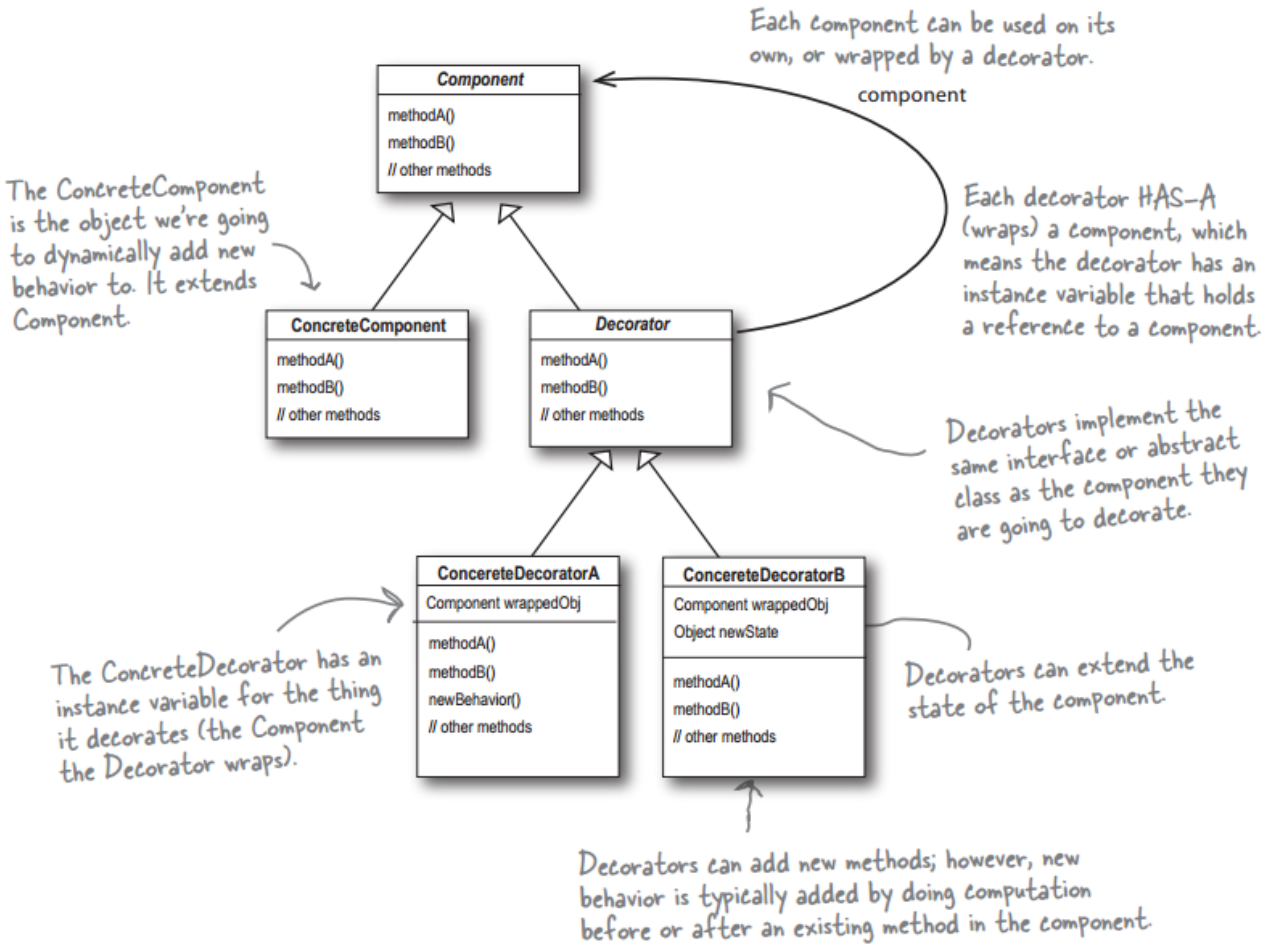

Figure 4.1: Decorator Pattern [7, p. 14]

4.2.1 Designing Filters

Filters are like *Decorators* in the JADE knowledge base. "A Decorator pattern adds additional functionality to an object dynamically" [7, p. 14]. A Decorator provides an alternative to sub-classing in java for adding functionality. A class diagram showing the design of a decorator pattern is illustrated in Figure 4.1.

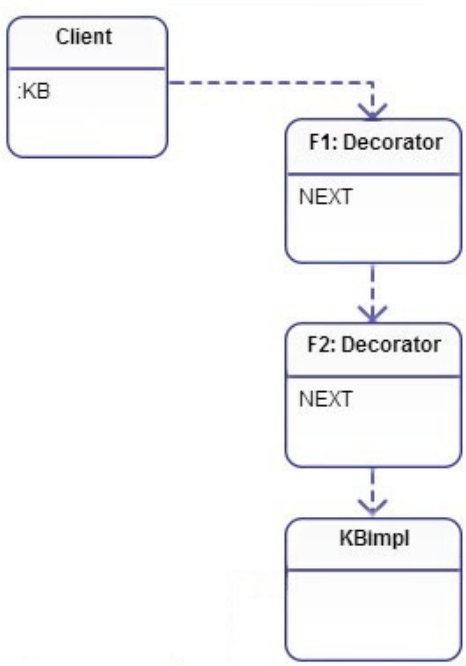

Figure 4.2: Object diagram for filters

The base class is sub-classed, not to inherit its behavior but to get its type. The behavior comes through the composition of decorators with the base class and other decorators. The filter design follows the object diagram shown in Figure 4.2. There could be a number of filters on top of the KBImpl. The formula is passed through the filters and then to KBImpl before generating the result of the query.

When querying the KB, the ontology filter is used to regenerate the query by modifying the identifiers in the query. The new queries are again passed to the KB. The overall process followed to produce the result for a client query is illustrated in Figure 4.3. The figure only gives a general case of querying a knowledge base with the ontology filter. First, the query is checked against the possible patterns `("?T", "(?P)", "(?P ?T)", "(?P ?T1 ?T2)")` and that the matches are simple identifiers that could be in the KB. If the query passes the check and a matching fact is found in the knowledge base, then `True` is returned as the output. If there is no matching fact then the query is modified. The modification of the

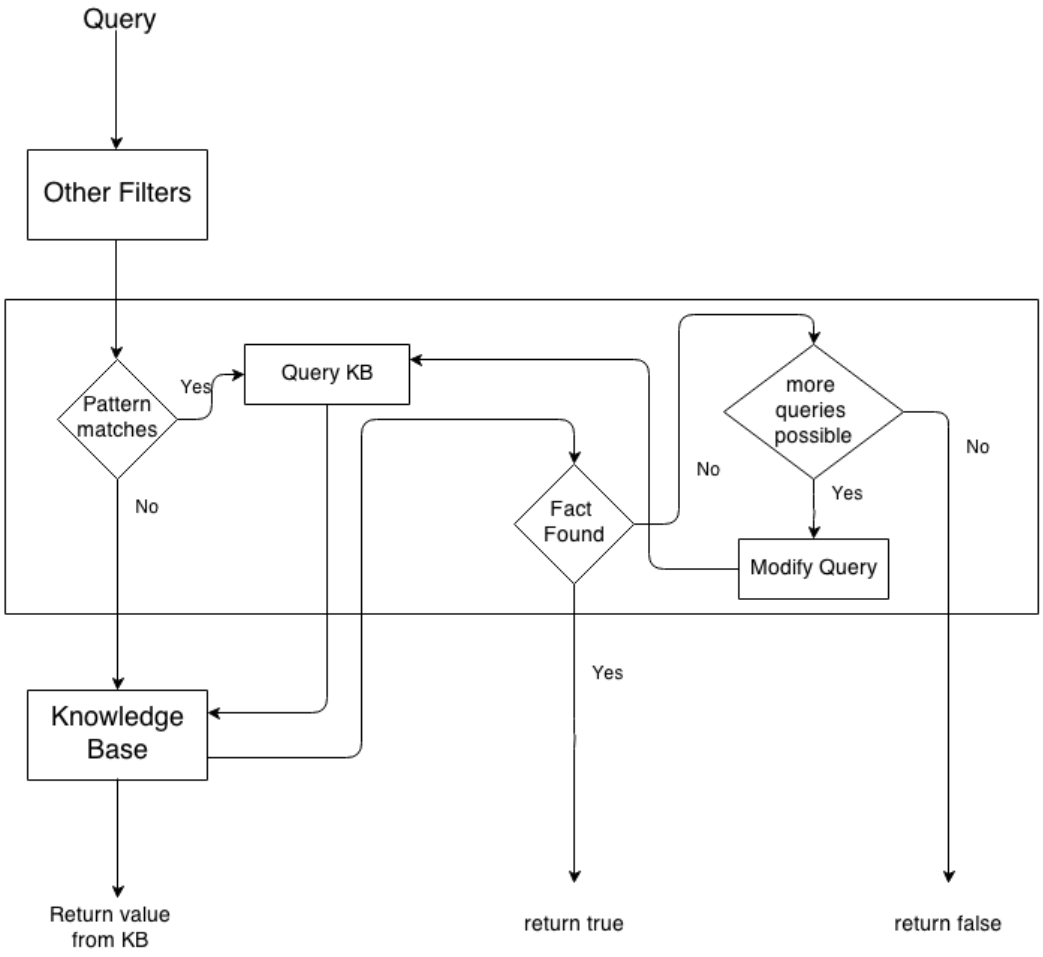

Figure 4.3: Query with ontology filters

query depends on the ontology. For instance, if the filter is queried with a query which has two identifiers, then all the parents (or children depending on type of subsumption) of the identifiers are generated from the ontology. For each identifier, parents (or children) are generated and added to a set. These sets are then added to a list of sets. This list is passed to an algorithm to generate all possible permutations of the query. The KB is then queried with all these queries. If a matching fact is found then `True` is returned as output. This process is recursively repeated until "TOP" is the parent (or "Nothing" is the child) of an identifier or a matching fact is found in the knowledge base. If after going through all the parents (or children) and all the queries, no matching fact is found, then `Nil` is returned as the output. The algorithm for the working of filters is detailed in Section 4.2.2.

4.2.2 Ontology Filter Algorithm

Algorithm 1 Algorithm for query search using ontology filter in CASA

```
1: CasaKB KB = Input to the algorithm
2: OWLOntology ont = Input to the algorithm
3: String query(query to be searched) = Input to the algorithm
4: result = KB.query(query)
5: if (result is True) then
6:     return true
7: else
8:     List<Set<String>> S = new ArrayList<Set<String>>()
9:     QueryResult result
10:    Generate an array called identifiers containing all the identifiers from query
11:    for i in identifiers do
12:        Set<String> A = new HashSet<String>()
13:        if ( i is present as a type in ont) then
14:            if (i follows reverse subsumption) then
15:                Add all children of i to A
16:            else
17:                Add all parents of i to A /* See Algorithm 2 */
18:            end if
19:            Add A to S
20:        end if
21:        Add i to A
22:        Add A to S
23:    end for
24:    Set<String> permutations
25:    Generate all permutations of S and add to permutations /* See Algorithm 3 */
26:    permutations.remove(query)
27:    for j in permutations do
28:        result = KB.query(j)
29:        if result == true then
30:            Add j to cache
31:            return true
32:        end if
33:    end for
34: end if
35: return Nil
```

Consider the Algorithm 1 given above. An ontology filter is given a *query* by an agent. An ontology filter is passed the knowledge base and the ontology. The variable *ont* holds the ontology. The variable KB holds the knowledge base. A list that contains sets of strings is initialized in line 8. An array called $identifiers$ is generated from the query in line 10. The query is first checked in the KB in line 4. If a matching fact is found, *true* is returned (lines

Algorithm 2 Algorithm for generating parents of an identifier

1: $identifier$: String /* Input*/
2: SI : Set<String>
3: Add $identifier$ to SI
4: Generate the immediate parents of $identifier$ and add it to an set of Strings called $parents$
5: **for** p in $parents$ **do**
6: Add p to SI
7: **end for**
8: return SI

5, 6), otherwise the else part is executed.

For each identifier in the array `identifiers` a set of strings A is initialized in line 12. If reverse subsumption is to be applied on the identifier then all the children of the identifier are generated from the ontology and are added to A. If normal subsumption is to be applied on the identifier then all the immediate parents of the identifier are generated from the ontology and are added to A (See Section 4.2.2.1). Also the identifier itself is added to A. So, for each identifier a set A is generated and added to S. A recursive algorithm is then applied on S in line 25 to generate all the permutations of the query and this is added to *permutations*. The knowledge base is re-queried with each of the permutations. If an exact fact is found then `true` is returned as output and the *query* is added to the cached knowledge base. This is repeated until all the parents or children are checked. After checking all the queries, if no match is found, `Nil` is returned as the output for the query.

4.2.2.1 Generating parents

Consider the Algorithm 2 above. The input for this algorithm is a string $identifier$ for which the parents are to be generated. An empty set of strings SI is initialized to null in line 2. This set will hold all the parents of the identifier. First of all, the $identifier$ is added to SI in line 3. All the immediate parents of $identifier$ are generated in line 4 and stored in a set of strings called *parents*. For each parent p in *parents*, p is added to SI. This is

Algorithm 3 Algo3: Algorithm for generating query permutations

```
1: sets : Sequence<Set<String>> /* Input*/
2: if Size of sets is 1 then
3:     return sets /* Terminating condition for only single element in the list sets */
4: end if
5: if Size of sets is 2 then
6:     Call Algorithm 4 with both sets as argument /* Terminating condition */
7:     returns the permutations over the elements of the two sets
8: else
9:     return = recursive call to Algo3 with parameter tail of sets
10: end if
```

implemented in lines 5, 6 and 7. Finally, SI is returned in line 8 which has all the generated parents of the identifier and the identifier itself. The algorithm for generating the children of an identifier is exactly similar to this algorithm. The only difference is that children of an identifier are generated instead of the parents.

4.2.2.2 Generating permutations recursively

Consider Algorithm 3 above. This algorithm takes a list (with each element a set of strings) *sets* as input in line 1 and outputs a set of strings (queries) containing a space delimited sequence of identifiers from the original *sets*. If the list only has one set then it is returned in line 3. Otherwise the algorithm is called again with the list without the first set as the arguments.

The terminating condition is when we only have two elements in the list. Then the Algorithm 4 is called (see section 4.2.2.3), which generates all the permutations for two sets.

4.2.2.3 Generating permutations for two sets

The algorithm 4 takes two sets of strings $set1$ and $set2$ as input in line 1 and 2. Each element from the first set is concatenated with each element of the second set and added to B. B is returned in line 9.

The next Section details the complexity analysis of the search using this algorithm.

Algorithm 4 Algorithm for generating permutations from two sets

```
1: set1 : Set<String> /* Input*/
2: set2 : Set<String> /* Input*/
3: B : Set<String>
4: for i in set1 do
5:     for j in set2 do
6:         Add (i+" "+j) to B
7:     end for
8: end for
9: return B
```

4.2.3 Complexity analysis of searching using the ontology filter algorithm

For this research the worst case time complexity of the ontology search is computed. The worst case complexity will be determined in terms of big-O notation.

4.2.3.1 Analysis

Suppose the query has n identifiers. For this analysis a bottom-up approach is being applied. Let us first compute the complexity of generating all the permutations from n sets using Algorithm 3 and Algorithm 4. Let's assume that each set has x elements then the complexity of generating permutations for 2 sets would be $O(x^2)$. If we have n sets then the complexity would be $O(x^n)$. This would be the overall complexity of generating all the permutations from a list of sets in the worst case.

Next let us calculate the complexity of generating the parents or children of the identifier using Algorithm 2. This depends on the height and width of the ontology lattice. Suppose each identifier has p elements (either parents or children). So the time taken for generating the parents (or children) of an identifier would be p. If we have n identifiers then overall complexity would be O(n*p).

Next from the Algorithm 1, steps 1 to 9 take a constant number of steps in the overall searching process. Step 10 takes n units of time to generate the array of identifiers. So the complexity of step 8 would be O(n). Steps 11 and 12 take constant number of steps. In step 15, all children (or ancestor) of the identifier are generated respectively using Algorithm 2.

So the complexity of generating the set of parents for each identifier will be O(np).
Again, steps 19 to 24 take a constant number of steps. In step 25 all the permutations are generated using Algorithm 3 and Algorithm 4. This algorithm takes $O(x^n)$ steps to execute in the worst case. The last for loop in line 27 performs recursion to check all the permutations in the knowledge base. This will then take $O(x^n)$ steps to execute (some multiplier of (x^n)).

The overall time complexity in worst case comes out to be O(n)+ O(n*p) + $O(x^n)$ + $O(x^n)$ + some constant. Since the filters are tested against small ontologies, the value of p is less then n^2 (means small number of children and parents for an identifier). If n is big, then the overall complexity comes to $O(x^n)$. This complexity will be compared with the actual time taken in Chapter 5.

4.3 Implementation

This Section will give the details about the implementation of the ontology filter in an agent based system called CASA.
CASA has been chosen for this research work due to the reasons explained in Section 2.1.4. OWL2 is the ontology language of the system. Details about OWL2 are given in Section 2.3.1.2 respectively.

The research was implemented on the HP PC and compatible machines in Microsoft Windows 7 environment. It was implemented using Java and Lisp languages. The Common Lisp implementation is ABCL [32]. This is implemented in Java and is platform independent. The filter was developed over eight month period, including design, implementation, testing, debugging and learning CASA. The filter code is approximately 500 lines. The largest part of time was taken to design and implement the test cases for the filter. The development initially focused on passing the queries directly to the knowledge base without using ontology

Figure 4.4: Asserting an fact to KB

filter. If there was no fact about the query in the knowledge base only then the query was passed through the ontology filter. The filter was developed iteratively meaning it was re-worked to add refinements.

4.3.1 Building and displaying the knowledge base

Each agent has a knowledge base which stores facts. An agent is able to assert a fact to the knowledge base using the Lisp function `KB.Assert`. `KB.Assert` takes a formula as input and asserts it to the knowledge base of the agent. Code for `KB.Assert` is given in A.1. For instance, if an agent wants to assert that "all animals have fur" in the KB, then the following command is run from the agent command line:

`(KB.assert "(hasfur animal)")`.

A screen-shot of this assertion to the KB in the CASA agent based system is illustrated in Figure 4.4.

Another important feature implemented was to display the facts already present in the knowledge base of an agent. `KB.Show` was implemented to display the knowledge from the knowledge base. The code for `KB.Show` is given in appendix A.2. From the previous example,

if the agent does a `(KB.Show)` after asserting `(KB.assert "(hasfur animal)")`, the output produced is displayed in Figure 4.4. The output displays the facts asserted i.e, `(hasfur animal)`. The numerical value is the total number of facts in the knowledge base of the agent.

4.3.2 Displaying agent ontologies

Each agent can load ontologies to be used in the filter. An agent could load more than one ontology. The ontologies are placed in files with .owl extension. After an agent is initialized, the loaded ontologies can be checked by using the `Ont.get` command. The `Ont.get` command lists all the agent ontologies and the location they are loaded from. The code for `Ont.get` is given in Appendix A.3. For our agent, running the `Ont.get` gave the following output:

Retrieving agent's default ontology...
Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
Prefix(rdf:=<http://www.w3.org/1999/02/22−rdf−syntax−ns#>)
Prefix(xml:=<http://www.w3.org/XML/1998/namespace>)
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix(rdfs:=<http://www.w3.org/2000/01/rdf−schema#>)

Ontology(<Fred.owl>
Import(<dataFiles/biology.owl>)
Import(<http://casa.cpsc.ucalgary.ca/ontologies/casa.owl>)
Import(<dataFiles/Predicates.owl>)
Annotation(rdfs:comment "CASA TransientAgent default ontology"^^xsd:string)

AnnotationAssertion(rdfs:comment <Fred> "Loaded from IRI: file:/C:/Users/Baljeet/workspace4/
CPSC671workspace/CASA/dataFiles/Fred.owl"@en)
AnnotationAssertion(rdfs:comment <file:/C:/Users/Baljeet/workspace4/CPSC671workspace/CASA/
dataFiles/Fred.owl> "Loaded from IRI: file:/C:/Users/Baljeet/workspace4/CPSC671workspace/
CASA/dataFiles/Fred.owl"@en))

This output gives information that the agent has been loaded with the biology, casa and Predicates ontologies. To retrieve the complete ontology, one uses a variation of `Ont.get` i.e, `Ont.get :imports T`. This variation will display all of the agent's ontologies as well as the imported ontologies. For these owl ontology files see Appendix B.1, B.2 and B.3.

4.3.3 Ontology Search Filter

An agent has a default knowledge base and some loaded ontologies. An agent should use the knowledge from the ontologies on top of the knowledge from the knowledge base to produce results for queries. A query can consist of several identifiers. The overall procedure of the search is listed in the next paragraph.

First of all the query is sent to the knowledge base to find an exact matching fact by executing the `Kb.query` command as in line 4 from Algorithm 1 in Section 4.2.2. If an exact match is found then `True` is returned as the result of the query (lines 5 and 6 from Algorithm 1 in Section 4.2.2). Otherwise the query is passed to the `searchOntology` method. Code for `Kb.query` and `searchOntology` are given in Appendix A.4 and A.5 respectively.

In the `searchOntology` method, first of all the query is split to find all the identifiers in the query (line 10 from Algorithm 1 in Section 4.2.2). For each identifier in the query, all the parents (or children) of the identifier are generated from the ontology using `recursiveParent` (or `recursiveChildren`) function. This function returns a set of all the parents of the identifier . For each identifier, the generated set is added to the list of sets. This list is then passed to the `recursivePermutations` method to generate all the possible permutations for the query (lines 25 from Algorithm 1 in Section 4.2.2 and Algorithm 3 in Section 4.2.2.2). Each of the permutation generated is then queried in the knowledge base to find a matching fact. This procedure is repeated until all parents and permutations have been checked. If at any point a matching fact is found in the knowledge base, `True` is returned as the result of the query and the fact is added to the cached knowledge base and the algorithm stops, otherwise `Nil` is returned as the result of the query. Code for `recursiveParent`, `recursiveChildren` and `recursivePermutations` are given in Appendix A.7, A.8 and A.9 respectively.

Consider an agent's knowledge base is asserted with the fact `(hasfur mammal)`. If the agent is queried `(hasskin dog)`. The algorithm will generate a set of all identifiers i.e. `hasskin, dog`. For each of these identifiers, a set is generated containing the parents (or

children) of the identifiers. The sets generated for these identifiers are {`hasskin`, `hasfur`}, {`dog`, `mammal`, `vertibrate`}. These sets are generated using the `Predicates` and `biology` ontologies. Both of these sets are passed to the `generatePermuatations` method to generate the permutations between them. The set of permutations is {`(hasskin dog)`, `(hasskin mammal)`, `(hasskin vertibrate)`, `(hasfur dog)`, `(hasfur mammal)`, `(hasfur vertibrate)`}. The agent is then queried over each permutation generated. Since a matching fact `(hasfur mammal)` is in the KB, so true would be returned as the result of the query and the cached knowledge base will be asserted with the fact `(hasskin dog)`.

The next sections discuss the various tests implemented for testing the ontology filter. For all these test cases an agent is loaded with the `Myontology.owl` ontology file. `Myontology` imports the `biology` and the `Predicates` ontologies.

4.3.3.1 Tests implementation

A testing program was coded which implements all the test cases from the requirements chapter. The code for the test program is given in Appendix C.1. In the test program the agent's knowledge base is asserted with `hasfur`, `(hasfur)`, `(hasfur mammal)` and `(hasfur mammal mammal)`.

4.3.4 Implementation with caching of the knowledge base

None of the requirements given in Section 4.3.3 use cached knowledge of the knowledge base. In all the test cases, if `True` is the result, then the query is added to a cache knowledge base. If the filters are to be implemented using caching, then the query is checked in both the original knowledge base and the cached knowledge base. The search times are recorded to compare them against the system having no cached knowledge.

4.4 Summary

This chapter describes the design and implementation of ontology filters in the CASA agent based system. An algorithm for the filter is presented and its complexity analysis is explained. In the implementation, different test cases are explained. A modified algorithm with caching of the knowledge is also presented. The details about the results and their analysis are given in the next Chapter.

Chapter 5

Results

5.1 Introduction

The previous chapter focused on the details about the design and implementation of the ontology filter in the CASA agent based system. This chapter describes the results of applying the ontology filter in CASA. The description includes references back to the design and implementation in the previous chapter.

Section 5.2 will detail the results of applying the ontology filter to the CASA agent based system. In this section the results will be compared against the system having no filters. In section 5.3, the results detail the time taken by the agents in searching with and without caching in the knowledge base. Section 5.4 gives the results of tests on individuals. Section 5.5 gives the results of applying the ontology filter on some complex queries. Section 5.6 will compare the actual time with the worst case time complexity.

5.2 Results of having ontology filters in CASA system

Ontology filter was developed and tests were performed to check their effect on the query results. Tests were performed to check subsumption of zero, one and two argument queries. After performing these tests, filters were removed and tests were performed again. For these tests the agents were loaded with biology (Appendix B.1) and predicates (Appendix B.2) ontologies.

Table 5.1: Results for subsumption of zero-argument with `hasfur` and `(hasfur)` in the KB

Test case number	Query	Result with filter	Result without filter
1	hasskin	F	F
2	hasfur	T	T
3	hashair	T	F
4	hastailhairs	T	F
5	(hasskin)	T	F
6	(hasfur)	T	T
7	(hashair)	F	F
8	(hastailhairs)	F	F

5.2.1 Zero argument test results

The agent's knowledge base was asserted with the fact `(hasfur)` and `hasfur`. This agent was given the queries listed in table 5.1 and the results for those queries with and without filter were recorded. These results are given in table 5.1. These results are in accordance to the requirements listed in Section 3.3.1.

5.2.1.1 With Filter

In the cases numbered form 1 to 4 the ontology filter follows normal subsumption. This means that parents are generated and checked for in the knowledge base. In test case number 1, the query searched for is `hasskin`. The agent then finds that there is no fact about `hasskin` or its parents in the knowledge base. So it returns false as the output. In test case number 2, the query is `hasfur` and agent exactly finds it in the knowledge base and returns `true` as the output. In test case number 3, the query is `hashair`. The agent finds that parent of `hashair` is in the knowledge base. So `true` is returned as the output. In the test case number 4, the query is `hastailhairs`. The agent finds that an ancestor of `hastailhairs` is in the knowledge base. So `true` is returned as the output. In the cases numbered form 5 to 8 the ontology filter follows reverse subsumption. This means that instead of generating parents, children are generated and checked for in the knowledge base. In test case number 5, the query searched for is `(hasskin)`. The agent then finds that there is a fact about `(hasskin)` or its child in the knowledge base. So it returns true as the output. In test case number 6,

the query is `(hasfur)` and agent exactly finds it in the knowledge base and returns `true` as the output. In test case number 7, the query is `(hashair)`. The agent finds that no child of `hashair` is in the knowledge base. So `false` is returned as the output. In the test case number 8, the query is `(hastailhairs)`. The agent finds that no child for `hastailhairs` is in the knowledge base. So `false` is returned as the output.

5.2.1.2 Without Filter

In test case number 2 and 6, an exact fact is found in the knowledge base. So `true` is returned as the output. For all the other test cases no matching fact is found in the knowledge base, as a result `false` is returned as the output.

5.2.2 One argument test results

An agent was first initialized. The agent's knowledge base was asserted with the fact `(hasfur mammal)`. This agent was given the queries listed in table 5.2 and the results for those queries with and without filter were recorded. These results are given in table 5.2. These results are in accordance to the requirements listed in Section 3.3.2.

5.2.2.1 With Filter

In test case number 1, the query searched is `(hasfur animal)`. The agent goes through all the possible queries which could be generated using the ontology filter and none of them is found in the knowledge base. So it returns false as the output. In test case number 2, the query is `(hasfur mammal)` and agent exactly finds it in the knowledge base and returns `true` as the output. In test case number 3, the query is `(hasfur dog)`. The agent finds that a proper parent `mammal` for `dog` has a fact in the knowledge base. So `true` is returned as the output. In test case number 4, the query is `(hasfur goldenretriever)`. The agent finds that an ancestor `mammal` for `goldenretriever` has a fact in the knowledge base. So `true` is returned as the output. In these test cases the first argument follows the reverse subsumption and the second argument follows forward subsumption. In test case

Table 5.2: Results for subsumption of one-argument with `(hasfur mammal)` in the KB

Test case number	Query	Result with filter	Result without filter
1	(hasfur animal)	F	F
2	(hasfur mammal)	T	T
3	(hasfur dog)	T	F
4	(hasfur goldenretriever)	T	F
5	(hasskin animal)	F	F
6	(hasskin mammal)	T	F
7	(hasskin dog)	T	F
8	(hasskin goldenretriever)	T	F
9	(hashair animal)	F	F
10	(hashair mammal)	F	F
11	(hashair dog)	F	F
12	(hashair goldenretriever)	F	F
13	(hastailhairs animal)	F	F
14	(hastailhairs mammal)	F	F
15	(hastailhairs dog)	F	F
16	(hastailhairs goldenretriever)	F	F

number 5 even after substituting the children and parent of arguments no fact is found in the knowledge base. So `false` is returned as the result. For test cases number from 6 to 8 agent finds matching facts in the knowledge base. So for these cases `true` is returned as the output. For test cases number from 9 to 16, the agent finds that the knowledge base has no fact about the parent or child. So for these cases `false` is returned as the output.

5.2.2.2 Without Filter

In test case number 2, an exact fact is found in the knowledge base. So `true` is returned as the output. For all the other test cases no matching fact is found in the knowledge base, as a result `false` is returned as the output.

5.2.3 Two argument test results

An agent was first initialized. The agent's knowledge base was asserted with the fact `(hasfur mammal mammal)`. This agent was given the queries listed in table C.1 in Appendix C.3 and the results for those queries with and without filter were recorded. These results are given

in Table C.1 in Appendix C.3. These results are in accordance to the requirements listed in Section 3.3.3.

5.2.3.1 With Filter

In these test cases the first argument follows reverse subsumption and the other two arguments follow normal subsumption. In test case number 1, the query searched is `(hasfur mammal animal)`. The agent then finds no fact about the possible queries in the knowledge base. So it returns false as the output. In test case number 2, the query is `(hasfur mammal mammal)` and agent exactly finds it in the knowledge base and returns `true` as the output. In test case number 3, the query is `(hasfur mammal dog)`. The agent finds that a proper parent `mammal` for `dog` has a fact in the knowledge base. So `true` is returned as the output. In the test case number 4, the query is `(hasfur mammal goldenretriever)`. The agent finds that an ancestor `mammal` for `goldenretriever` has a fact in the knowledge base. So `true` is returned as the output. For test cases number from 5 to 9 , the agent finds no fact in the knowledge base. So for these cases `false` is returned as the output. For all the other cases the results follow the same pattern as the previous results.

5.2.3.2 Without Filter

In test case number 2, an exact fact is found in the knowledge base. So `true` is returned as the output. For all the other test cases no matching fact is found in the knowledge base, as a result `false` is returned as the output.

5.3 Time taken for searching in the KB

In section 5.2 all the test cases are detailed with the outputs. Now, the time taken by all the cases will be explained and a graphical analysis will be performed. The system was tested with and without caching of the knowledge in knowledge base. For these test cases, labels are used. 0 is the label for *in the KB*, 1 is the label for *proper parent in the KB*, 2 is the

Table 5.3: Time (microseconds) for zero argument tests with `(hasfur)` in the KB

Query	Result	Label	Average time without caching	Average time with caching
(hasskin)	T	-1	5260	7350
(hasfur)	T	0	1720	1720
(hashair)	F	1	5670	5580
(hastailhairs)	F	2	2640	2670

label for *ancestor in the KB* and -1 is the label for *child in the KB*.

5.3.1 Time for zero argument test results

The agent's knowledge base was asserted with the fact `(hasfur)`. This agent was given the queries listed in table 5.3 and the results for those queries and time was recorded. This was repeated 10 times and the average time was computed. The average time is given in table 5.3. The same process was repeated for the system with caching in the knowledge base.

A graph showing the average time (in microseconds) and label columns is given in figure 5.1. The horizontal axis has the label for different test cases. The vertical axis has the average time. The dotted line is the time when the system has caching and the solid line is the time without caching. Clearly there is a decrease in time (for most cases) when caching is present. The reason for this is that all the queries which are true are being added to a cached knowledge base. Once an agent is queried again it checks the cached knowledge base and a lot more facts are found in it. So the search time is greatly decreased. The reason for the time in test cases with labels 1 and 2 being close to double that of test case with label 0 is that in those test cases two different knowledge bases are checked for an matching fact.

For the non-caching cases, the time taken by the test case with label 0 is the smallest as it is present in the KB. The test case with label 2 takes less time as compared to the test case with label 1. This is because for the test case with label 2 no proper child of the identifier is in the ontology and for the test case with label 1 a child of the identifier is in the ontology. Since there is two levels of recursion in the test case with label 1, it takes more time. The test case with label -1 has a time higher then other cases. This is because a child

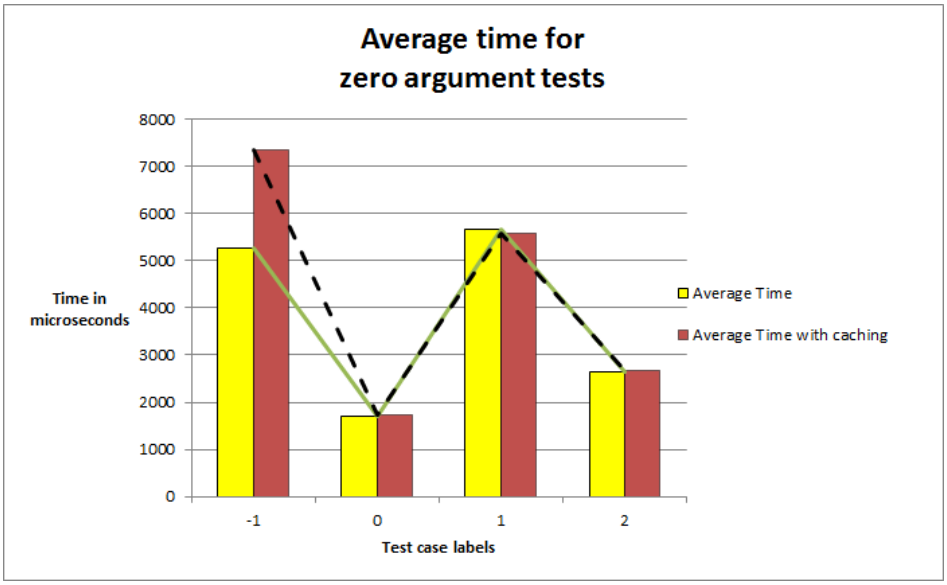

Figure 5.1: Average time(microseconds) of 10 runs for zero argument test cases

of the identifier is in the KB and true is the result for that case.

5.3.2 Time (microseconds) for one argument test results

The agent's knowledge base was asserted with the fact `(hasfur mammal)`. This agent was given the queries listed in table 5.4 and the results for those queries and time was recorded. This was repeated 10 times and the average time was computed. The average time is given in table 5.4. The same process was repeated for the system with caching in the knowledge base.

A graph showing the average time (in microseconds) and label columns is given in Figure 5.2. The horizontal axis has the labels for different test cases. The vertical axis has the average time. The dotted line is the time when the system has caching and the solid line is the time without caching. The reason for less time taken in some test cases (with caching present) is because more facts are there in the knowledge base in the case of caching. The time taken is more when the query has identifiers having a large number of parents and children in the ontology. This is because the agent has to check all the parents and childeren before generating the result. Also, when the system has caching there are some high times

Table 5.4: Time for one argument tests with `(hasfur mammal)` in the KB

Query	Result	Label	Average time	Average time with caching
(hasskin animal)	F	-1-1	11800	11600
(hasskin mammal)	T	-10	20800	20500
(hasskin dog)	T	-11	42400	40500
(hasskin goldenretriever)	T	-12	37600	28900
(hasfur animal)	F	0-1	3780	5340
(hasfur mammal)	T	00	635	598
(hasfur dog)	T	01	25500	23300
(hasfur goldenretriever)	T	02	58400	33800
(hashair animal)	F	1-1	2390	1940
(hashair mammal)	F	10	7040	5520
(hashair dog)	F	11	13500	10600
(hashair goldenretriever)	F	12	16600	24200
(hastailhairs animal)	F	2-1	830	854
(hastailhairs mammal)	F	20	1480	1740
(hastailhairs dog)	F	21	2070	2540
(hastailhairs goldenretriever)	F	22	2680	3460

for searching. This is because no fact is found in the either of the knowledge bases in these cases.

For the non-caching cases, the time taken by the test case number with label 00 is the smallest as it is present in the knowledge base. Some test cases have intermediate time as the search stops as soon as the "TOP" is found as the parent of the identifier or "Nothing" is found as the child of an identifier. The test cases numbered from 9 to 16 in table 5.4 follow the same trend. In all the cases, when an ancestor of an identifier is in the knowledge base, time taken for search is more as compared to when a proper parent is in the knowledge base.

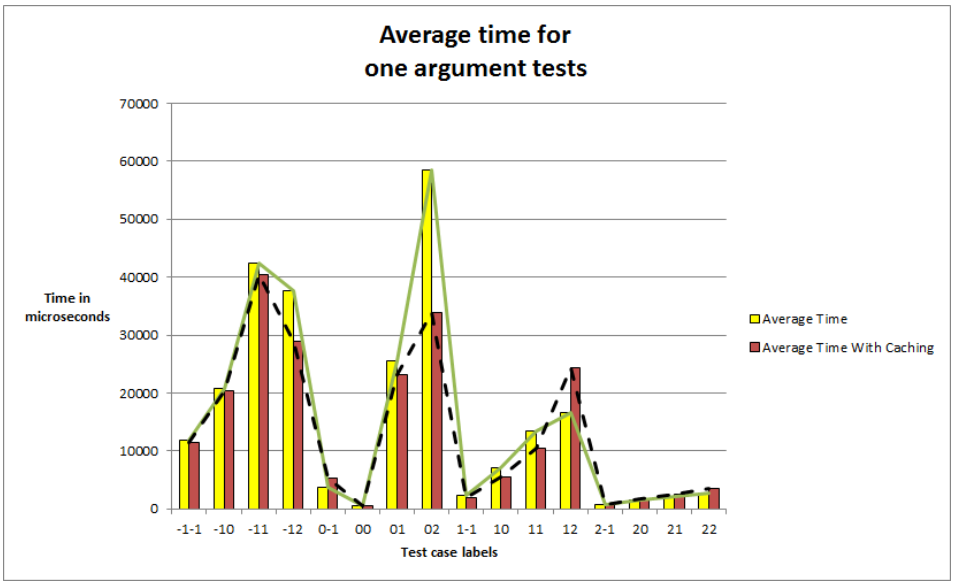

Figure 5.2: Average time(microseconds) of 10 runs for one argument test cases

5.3.3 Time for two argument test results

The agent's knowledge base was asserted with the fact `(hasfur mammal mammal)`. This agent was given the queries listed in table 5.5 and the results for those queries and time was recorded. This was repeated 10 times and the average time was computed. The average time is given in table 5.5. The same process was repeated for the system with caching in the knowledge base.

A graph showing the average time (in microseconds) and label columns is given in Figure 5.3. The horizontal axis has the labels for different test cases. The vertical axis has the average time. The dotted line is the time when the system has caching and the solid line is the time without caching. It can be clearly seen that time for most of the test cases is similar with and without caching. The reason for that is most of the results returned are `false`. So many fewer facts are added to the cached knowledge base. As a result there are very few facts in either of the knowledge bases to match against. As a result, the time for searching is comparable.

For the non-caching cases, the time taken by the test case number 22 is the smallest as it is present in the knowledge base. All the other cases follow the same pattern as for one-

Table 5.5: Time (microseconds) for two argument tests with `(hasfur mammal mammal)` in the KB

Query	Result	Label	Average time	Average time with caching
(hasskin animal animal)	F	-1-1-1	3720	4790
(hasskin animal mammal)	F	-1-10	16100	16400
(hasskin animal dog)	F	-1-11	44900	42400
(hasskin animal goldenretriever)	F	-1-12	99100	79700
(hasskin mammal animal)	F	-10-1	8610	7980
(hasskin mammal mammal)	T	-100	9030	9260
(hasskin mammal dog)	T	-101	10400	10500
(hasskin mammal goldenretriever)	T	-102	40600	32200
(hasskin dog animal)	T	-11-1	21500	22100
(hasskin dog mammal)	T	-110	7190	7550
(hasskin dog dog)	T	-111	9700	9730
(hasskin dog goldenretriever)	T	-112	9680	9930
(hasskin goldenretriever animal)	T	-12-1	54600	54500
(hasskin goldenretriever mammal)	T	-120	150500	152600
(hasskin goldenretriever dog)	T	-121	8970	9330
(hasskin goldenretriever goldenretriever)	T	-122	705000	702000
(hasfur animal animal)	F	0-1-1	850	880
(hasfur animal mammal)	F	0-10	3490	3590
(hasfur animal dog)	F	0-11	9170	9460
(hasfur animal goldenretriever)	F	0-12	19400	19400
(hasfur mammal animal)	F	00-1	3440	3570
(hasfur mammal mammal)	T	000	169	196
.....	...	...		
.....	...	...		
.....	...	...		
(hastailhairs goldenretriever mammal)	F	220	6190	6180
(hastailhairs goldenretriever dog)	F	221	19500	19400
(hastailhairs goldenretriever goldenretriever)	F	222	48300	48900

argument and zero-argument results i.e., queries having a proper parent in the knowledge base have less time as compared to queries having an ancestor in the knowledge base.

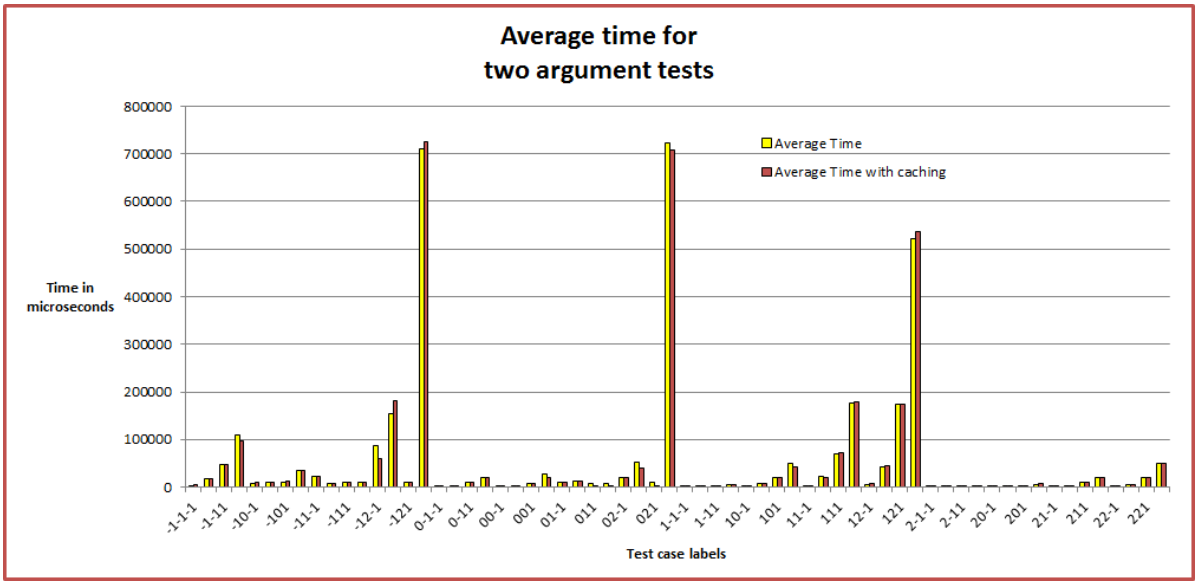

Figure 5.3: Average time(microseconds) of 10 runs for two argument test cases

5.4 Individuals test results

Consider the biology ontology from Appendix B.1. For these tests, the knowledge base is asserted with `mammal`, `(mammal)`, `(hasfur mammal)` and `(hasfur mammal mammal)`. `rover` is an iindividual of `goldenretriever` class. The agent was tested for the queries as given in table 5.6. The results of these tests are listed in the column Result.

If the query is made about `rover` like `?rover` then the result should be `true` and if the query is made about `tom` like `?tom` the result should be `nil` as it is in the knowledge base. In test case number 3 and 5 , the query returned `True` as `rover` is an instance of `mammal` and in test case number 4 and 6, the query returned `False` as `Tom` is not in the KB or ontology.

Table 5.6: Individual test results with `mammal`, `(mammal)`, `(hasfur mammal)` and `(hasfur mammal mammal)` in the KB

Test case number	Query	Result
1	rover	T
2	tom	F
3	(hasskin rover)	T
4	(hasskin tom)	F
5	(hasskin rover rover)	T
6	(hasskin rover tom)	F

Table 5.7: Results for not filter with `(hasfur)`, `(hasfur mammal)`, `(hasfur mammal mammal)` in the KB

Test case number	Query	Result
1	(! (hasfur dog))	F
2	(! (hasskin animal))	T

5.5 Other complex test cases

The ontology filter was also tested with other filters like *and*, *or*, *not*. Some of the test queries for these and their results are given in the following sections. For performing tests, the agent's knowledge base was asserted with the facts `(hasfur)`, `(hasfur mammal)`, `(hasfur mammal mammal)`. For the complete test file see Appendix C.2.

5.5.1 Not filter tests

The agent was tested for the queries as given in table 5.7.

In test case number 1, query returned `True` and the `not filter` inverted it to `False`. In test case number 2, query returned `False` and the `not filter` inverted it to `True`.

5.5.2 And filter tests

The agent was tested for the queries as given in table 5.8.

In test case number 1, both the queries returned `True`. So the overall result of `and filter` on it was `True`. In the test case number 2, 3 and 4, one or more of the queries had `False` as the result. So the overall result was `False`.

Table 5.8: Results for and filter with `(hasfur)`, `(hasfur mammal)`, `(hasfur mammal mammal)` in the KB

Test case number	Query	Result
1	(and (hasfur dog) (hasfur mammal))	T
2	(and (hasfur dog) (hasskin animal))	F
3	(and (hasskin animal) (hasfur dog))	F
4	(and (hasskin animal) (hashair mammal))	F

Table 5.9: Results for or filter with `(hasfur)`, `(hasfur mammal)`, `(hasfur mammal mammal)` in the KB

Test case number	Query	Result
1	(or (hasfur dog) (hasfur mammal))	T
2	(or (hasfur dog) (hasskin animal))	T
3	(or (hasskin animal) (hasfur dog))	T
4	(or (hasskin animal) (hashair mammal))	F

5.5.3 Or filter tests

The agent was tested for the queries as given in table 5.9.

In test case number 4, both the queries returned `False`. So the overall result of `or filter` on it was `False`. In the test case number 2, 3 and 4, one or more of the queries had `True` as the result. So the overall result was `True`.

5.6 Actual time against worst case time complexity

Consider the different test cases discussed in the previous sections. The theoretical time complexity of searching using the ontology filter from Section 4.2.3 is $O(x^n)$. Here x is the number of ancestors (or decendants) of an identifier and n is the number of identifiers in the query. The actual times from the test cases for different values of x and n are given in table 5.10.

Table 5.10: Actual test results

Number of parent (x)	Number of identifiers (y)	Time (in microseconds)
0	1	150
0	2	235
0	3	169
1	1	4520
1	2	25500
1	3	66600
2	1	4590
2	2	58400
2	3	705000

A graph plotting each point from this table against an graph for $O(x^n)$ (more specifically

$z=7x^{2.1y}$) is given in Figure 5.4. The graph for $z=7x^{2.1y}$ is a surface. The x-axis of the graph represents the number of identifiers, the y-axis represents the number of parents or children and the z-axis represents the time taken (scaled down by a factor of 1000). There are three lines in the graph. For the first line x is fixed to 0 and for the second line x is fixed to 1 and for the third line x is fixed to 2. For each of these values of x, the number of identifiers (n) is varied from 1 to 3 and the time is noted. Clearly, most of the data points lay on the surface. So the actual time is in accordance to the theoretical time for the ontology filter.

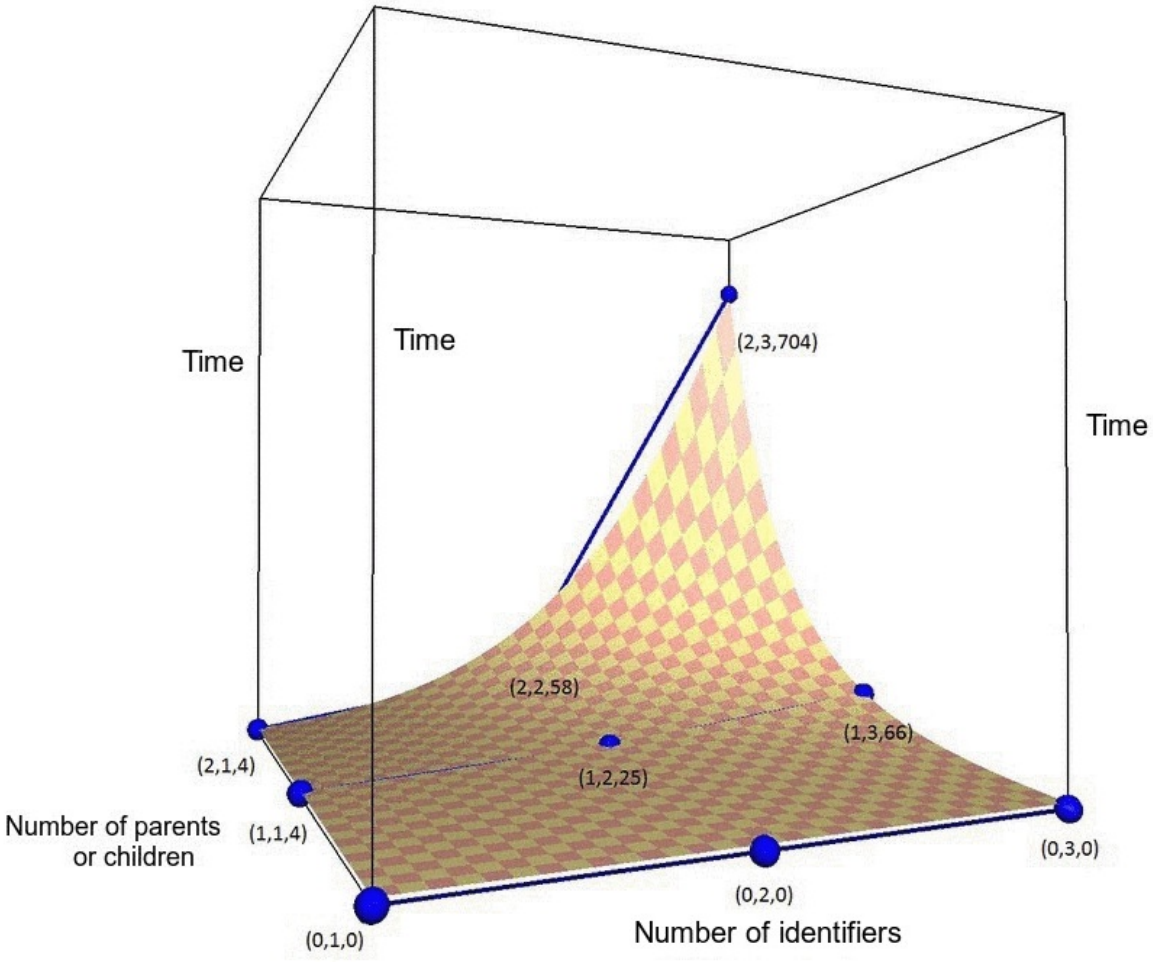

Figure 5.4: Actual time against theoretical time complexity

5.7 Summary

This chapter describes the results of an ontology filter on the CASA agent based system. The results are presented to see the effect of filters with and without the caching in the knowledge base. Times for searching in the system are plotted in graphs to do analysis. The graphs show that caching of knowledge in the knowledge base reduces the query search time in many cases and increases the search time if no matching fact is found in either of the knowledge bases. After that the actual time taken for queries are compared against the worst case time and found an reasonable match. Overall the results show an improvement in the query answering. This means that now the agent no longer produces false negative answers with the benefit of ontological knowledge, but takes a lot of time to produce the results.

Chapter 6

Conclusions

6.1 Introduction

This thesis presents an approach for adding and testing the ontology filter in the CASA agent based system. The focus of the thesis is on testing the correctness and efficiency of the search using the ontology filter on top of the knowledge base. Most of the research done towards this thesis provides insights about using an ontology filter in a query search. Chapter 1 described the motivation and the goals of this research. In Chapter 2 an overview of various agent based systems, logics, ontologies and knowledge bases was provided for better understanding of this field. In Chapter 3, various requirements for performing this research were presented. It included system and testing requirements. Designing and implementation of the ontology filter and its various components were detailed in Chapter 4. Chapter 4 also included the implementation of various test cases for this research. Chapter 5 focused on the outcomes of the various test cases. This included the analysis of the output to the queries and the time taken for the queries. Time taken was noted both for system with and without caching of knowledge.

6.2 Conclusions from results

The results in the previous chapter gave a lot of information about the ontology filter in the CASA system.

First of all, the results are more accurate when ontology filter is used. This means adding the ontology filter to the CASA system makes answers closer to the way humans will answer queries. If the queries are answered using the facts present in the knowledge

base (only knowledge from the KB), then only exact facts are known to the agent and very seldom `true` is returned. This makes the knowledge base really rigid as it is just doing pattern matching while answering the queries. Once the ontology filter is added, then the ontological knowledge gets added to the knowledge from the knowledge base and the results are much more accurate.

The results also checked the time taken by the agent to answer the query with the ontology filter. The results are accurate but the search time for the queries is dramatically increased. This is because the query is passed through the ontology filter which generates all the possible queries from the identifiers in the original query. Then all of these new generated queries are checked in the knowledge base. This makes the search time longer. To improve the search time, caching is introduced in the system. In caching a cache knowledge base is created. When a query is answered as `true`, it is added to the cache knowledge base. Using caching the query is now checked for a matching fact in both the knowledge base and the cache. If the searched fact is present in the cache, the system takes less time for producing the result. But if the searched fact is not in cache and is not even in KB then the search times were large. So using cache was a time trade off.

The theoretical time complexity in the worst case for the ontology search comes out to be O(n)+ O(n*p) + O(x^n) + O(x^n) + some constant. See Section 4.2.3 for details. If n is big, then the overall complexity comes to O(x^n). Here x is the number of parents (or children) of an identifier and n is the number of identifiers in the query. This complexity is compared with the actual time taken in the results chapter. An graph plotting the actual times of search show that most of the data points lay close to the surface $z=7x^{2.1y}$. So the actual time is in accordance to the theoretical time for the test cases. Overall, the results of adding the ontology filter to the CASA system makes the query answering accurate but with increased search time. It also removes the rigidness of the JADE semantic knowledge base. So ontological information should be used in systems which have rigid knowledge base.

Otherwise using ontological information increases the search time.

6.3 Contributions

The first contribution of this thesis is providing an overview about various multi-agent systems and their features. This was done by conducting a literature review to search for relevant research papers. This also helped in understanding about ontologies, knowledge bases and logics. This helped to understand the rigid nature of the knowledge base in the CASA system. This also answered our first research question aimed at understanding the way the queries were answered in the JADE Semantic Extension knowledge base.

The second research question focused on using the ontologies on top of the knowledge from the knowledge base. In alignment with this, the ontology filter was developed, which is the second contribution of this thesis. The ontology filter adds the knowledge from the ontologies on top of the knowledge from the knowledge base. The algorithm for the search using the ontology filter was discussed in Chapter 4. The ontology filter was designed with and without caching of the knowledge in the knowledge base.

The third contribution of this thesis is the improvement in the results of queries using the ontology filter. The tests were performed with and without caching of knowledge in the knowledge base. The results were more accurate using the ontological knowledge. The search time was increased when using the ontology filter. The research also had some limitations which are discussed in next section. This laid the foundation of the final contribution of this thesis i.e, providing the foundation for future research activities in this field.

6.4 Limitations of the research work

Consider a case when we have three types in the ontology as A, B and C. A subsumes B and B subsumes C. Suppose the knowledge base has the following facts:

```
(P A)
```

`not(P B)`

If the agent is then queried as `?(P C)`, the agent is not sure which knowledge to use. If the agent looks at the knowledge base then the result of this query is `false`. This is because no fact about C is in the knowledge base. If it uses the knowledge from the ontology, then the system outputs `true` as the result, which is not correct. The output generated is `true` because the filters do not match `not(P B)` and the knowledge base takes `not(P B)` to be a retraction instead of a new piece of knowledge. So it finds that a parent A is in the knowledge base of C and outputs `true`. While the ontology filter can properly handle the expressions with negation, it cannot currently handle negative knowledge in the knowledge base.

Another limitation is the increased time complexity. Adding the ontology filter to the CASA system makes the query answering accurate but with increased search time. The search time is less when the number of identifiers in the query is small and each identifier has small number of parents (or children), otherwise the search time is longer. Another case when the search time is less is when the cached knowledge base has the exact fact being queried.

6.5 Future Work

Adding the ontology filter on top of the knowledge base in the CASA agent based system is a new field in CASA. So this has several research perspectives in which work can be extended. These directions could not be explored within the scope of this thesis, so these are proposed as future work.

The first direction this research can be extended is to find a new way of handling the retraction vs negative knowledge in the knowledge base. This would include finding a way to handle the limitations from the previous section.

The second direction is to improve the time taken for searching the queries using an ontology filter. In this research the size of the ontologies against which tests were performed

was quite small. The filter should be tested against bigger ontologies to more accurately determine its search time. So, reducing the time complexity of the search would be another direction to extend this research.

Bibliography

[1] R. Kremer, "Casa user manual." http://pages.cpsc.ucalgary.ca/~kremer/CASA/doc/CasaUserManual.pdf, 2013. University of Calgary, Calgary, Canada.

[2] V. Pautret, "Jade semantics add-on programmer's guide." http://jade.tilab.com/doc/tutorials/SemanticsProgrammerGuide.pdf, 2006. France Telecom, Cedex, France.

[3] BBN Technologies, "Cougaar website." http://cougaar.org/, 2012. BBN Technologies, Cambridge, USA.

[4] W3C, "Owl 2 web ontology language document overview (second edition)." http://www.w3.org/TR/2012/REC-owl2-overview-20121211/. W3C OWL Working Group, December, 2012.

[5] M. Boris and P. Bijan, "Owl 2 web ontology language structural specification and functional-style syntax (second edition)." http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/. W3C OWL Working Group, December, 2012.

[6] British Computer Society Coventry Branch, "Artificial intelligence." http://coventry.bcs.org/resources/artil.htm, 2009. British Computer Society Coventry Branch, Coventry.

[7] F. Eric, F. Elisabeth, B. Bert, and S. Kathy, "Chapter 3: Head first design patterns," Sebastopol, USA: OREILLY, 2004.

[8] S. Staab, R. Studer, H.-P. Schnurr, and Y. Sure, "Knowledge processes and ontologies," *Intelligent Systems, IEEE*, vol. 16, no. 1, pp. 26–34, 2001.

[9] D. Kirsh, "Foundations of artificial intelligence," Cambridge, Massachusetts: The MIT Press, 1991.

[10] S. Ralph and R. George, "Principles of information systems." http://www.nelsonbrain.com/content/stair78292_0538478292_02.01_chapter01.pdf, 2012. Cengage Learning, Course Technology, Boston, MA.

[11] M. Uschold and M. Gruninger, "Ontologies: principles, methods and applications," *The Knowledge Engineering Review*, vol. 11, pp. 93–136, 6 1996.

[12] J. Denzinger, "Multi-agent systems projects." http://pages.cpsc.ucalgary.ca/~denzinge/projects/multi-agent.html, 2013. University of Calgary, Calgary, Canada.

[13] FIPA, "Fipa specifications website." http://www.fipa.org., last visited 2013. IEEE, Geneva, Switzerland.

[14] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa, "Jade: A software framework for developing multi-agent applications. lessons learned," *Information and Software Technology*, vol. 50, no. 1, pp. 10–21, 2008.

[15] A. Helsinger, M. Thome, and T. Wright, "Cougaar: a scalable, distributed multi-agent architecture," in *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, vol. 2, pp. 1910–1917, IEEE, 2004.

[16] M. Winikoff, "Jack intelligent agents: An industrial strength platform," in *Multi-Agent Programming*, pp. 175–193, Springer, 2005.

[17] M. Dastani, M. van Birna Riemsdijk, and J.-J. C. Meyer, "Programming multi-agent systems in 3apl," in *Multi-agent programming*, pp. 39–67, Springer, 2005.

[18] Open Source Initiative, "California public benefit corporation,lgpl license." http://www.opensource.org/licenses/lgpl-license.php, last modified 2013. Open Source Initiative, Palo Alto, California.

[19] FIPA, "Fipa sl content language specification." http://www.fipa.org./specs/fipa00008/SC00008I.pdf, 2002. IEEE, Geneva, Switzerland.

[20] J. H. Gallier, *Logic for computer science: foundations of automatic theorem proving*. Harper & Row Publishers, Inc., Philadelphia, USA, 1985.

[21] M. Krötzsch, F. Simancik, and I. Horrocks, "A description logic primer," 2012. University of Oxford, Wellington Square, UK.

[22] Indiana University Bloomington, "Predicate logic." http://csitiub.pbworks.com/w/file/fetch/72718706/Notes, 2013. PBworks Inc , San Mateo, California.

[23] J. McCarthy and P. Hayes, *Some philosophical problems from the standpoint of artificial intelligence*. Stanford University USA, 1968.

[24] A. Newell, "The knowledge level: presidential address," *AI magazine*, vol. 2, no. 2, p. 1, 1981.

[25] Stanford University, "Ontolingua." http://www.ksl.stanford.edu/software/ontolingua/, 2008 (last modified Tuesday, 22-Jan-2008 22:22:54 PST). Stanford University, CA, USA.

[26] A. Farquhar, R. Fikes, and J. Rice, "The ontolingua server: A tool for collaborative ontology construction," *International journal of human-computer studies*, vol. 46, no. 6, pp. 707–727, 1997.

[27] W3C, "Owl web ontology language document overview (second edition)." http://www.w3.org/TR/2004/REC-owl-guide-20040210/. MIT, February, 2004.

[28] R. W. Schuler, "The application of iso 10303-11 (the express language) in defining data models for software design and implementation.." http://www.geocities.ws/schulerrw/express/EXPRESS_data_modeling.pdf, April, 2001. Schuler.

[29] Wikipedia, "Express (data modeling language)." http://en.wikipedia.org/wiki/EXPRESS_(data_modeling_language), 2013. Wikimedia Foundation.

[30] Cycorp Inc., "Cycl: The cyc knowledge representation language." http://www.cyc.com/cyc/cycl, 2014. Cycorp Inc., Austin, USA.

[31] A. Martin and C. Luca, *A theory of objects*. Springer, New York, USA, 1998.

[32] Edgewall Software, "Armed bear common lisp (abcl)." http://abcl.org/, 2013. Edgewall Software.

[33] P. H. Rossi, M. W. Lipsey, and H. E. Freeman, "Evaluation: A systematic approach," Oaks, California: Sage, 2004.

Appendix A

Code of useful functions

A.1 Code snippets for Kb.Assert

```
/**
 * @param agent
 * @param params
 * @param ui
 * @param env
 * The formula is asserted in the knowledge base of the agent.
 */
private static final CasaLispOperator ASSERT =
   new CasaLispOperator("KB.ASSERT", "\"!Asserts the formula into the KB.\" "
      +" FORMULA \"@java.lang.String\" \"!The formula.\""
      , TransientAgent.class)
 {
   @Override
   public Status execute(TransientAgent agent, ParamsMap params, AgentUI ui, Environment env)
    {
           String formString = (String)params.getJavaObject("FORMULA");
           try
           {
            Formula form = agent.assert_(formString);
            return new StatusObject<Formula>(0,"",form);
           }
           catch (ParseException e)
           {
            return new Status(-1, "(ASSERT \""+formString+"\"): Bad Parse of expression.", e);
           }
         }
 }
```

A.2 Code snippets for KB.Show

```
/**
 * @param agent
 * @param params
 * @param ui
 * @param env
 * The method displays all the knowledge form the knowledge base of the agent.
 */
private static final CasaLispOperator KB_SHOW = new CasaLispOperator(
    "KB.SHOW",
    "\"!Dispalys the formulas from the KB.\" "
      + "&OPTIONAL PATTERN \"@java.lang.String\" \"!If specified, only expressions containing
          PATTERN will be displayed.\" "
      + "&KEY REGEX \"!Set to non-NIL to have PATTERN interpreted as a regular expression
          occuring in the expression displayed.\" "
```

```
      + "STRICT \"!Set to non-NIL to have PATTERN interpreted as a regular expression that must
          match the entire expression to be displayed.\" "
      + "FACTS \"!Set to non-NIL to return only a list of facts in the KB.\" "
      + "QUERIES \"!Set to non-NIL to return only a list of query filters on the KB.\" "
      + "ASSERTS \"!Set to non-NIL to return only a list of assert filters on the KB.\" "
      ,TransientAgent.class) {
  @Override
  public Status execute(TransientAgent agent, ParamsMap params, AgentUI ui,
    Environment env) {
   String pattern = (String) params.getJavaObject("PATTERN");
   boolean regex = params.getJavaObject("REGEX") != null;
   boolean strict = params.getJavaObject("STRICT") != null;
   boolean facts = params.getJavaObject("FACTS") != null;
   boolean queries = params.getJavaObject("QUERIES") != null;
   boolean asserts = params.getJavaObject("ASSERTS") != null;
   int count = 0;
   if (!facts && !queries && !asserts) {
    count += printKBexpressions(ui, pattern, agent.kBase.toString(), regex, strict);
   }
   else {
    if (facts) {
     count += printKBexpressions(ui, pattern, agent.kBase.toStringFacts(), regex, strict);
    }
    if (queries) {
     count += printKBexpressions(ui, pattern, agent.kBase.toStringQueryFilters(), regex, strict);
    }
    if (asserts) {
     count += printKBexpressions(ui, pattern, agent.kBase.toStringAssertFilters(), regex, strict);
    }
   }
   return new StatusObject<Integer>(0,count);
  }
 }
```

A.3 Code snippets for Ont.Get

```
/**
 * @param agent
 * @param params
 * @param ui
 * @param env
 * Retrieves ontology either from the shared memory or from a file of the same name. If none of :relation,
     :type, or :individual is specified the ontology is printed and returned.
 */
static final CasaLispOperator ONT_GET =
  new CasaLispOperator("ONT.GET", "\"!Retrieves ontology either from the shared memory or from a
      file of the same name ([name].ont.lisp). If none of :relation, :type, or :individual is specified the
      ontology is printed and returned.\" "
    +"&OPTIONAL NAME \"@java.lang.String\" \"!The name of the ontology.\" "
    +"&KEY VERBOSE \"!echo the command if verbose/=NIL\""
    +"INDIVIDUAL \"!print out a lisp description of an individual in the ontology, returning the
        individual object.\""
    +"TYPE \"!print out a lisp description of a type in the ontology, returning the type object.\""
    +"RELATION \"!print out a lisp description of a relation in the ontology, returning the relation
        object.\""
```

```
    +"(IMPORTS NIL) \"@java.lang.Boolean\" \"!print out the imported ontologies (ignored if
         INDIVIDUAL, TYPE, or RELATION is present).\""
    , TransientAgent.class, "GET-ONTOLOGY")
{
 @Override
 public Status execute(TransientAgent agent, ParamsMap params, AgentUI ui, Environment env) throws
       ControlTransfer {

  String ontName = null;

  if (params.containsKey("NAME"))
   ontName = ((String)params.getJavaObject("NAME"));

  Ontology ont = null;
  if (ontName==null) {
   try {
    if (ui!=null)
     ui.println("Retrieving agent's default ontology...");
    ont = (Ontology)agent.getOntology();
   } catch (Throwable e) {
    return new Status(-2,"Failed (ONT.GET) for current agent",e);
   }
  }
  else {
   params.put("ONTOLOGY", ontName, new SimpleString(ontName), false);
   ont = CASAUtil.findOntology(agent, params, ui, env);
  }
  if (ont==null) {
   return new Status(-1,"Ontology "+ontName+" does not exist");
  }

   String relName = (String)params.getJavaObject("RELATION");
   String typeName = (String)params.getJavaObject("TYPE");
   String indName = (String)params.getJavaObject("INDIVIDUAL");
   int c = 0;
   if (relName!=null) c |= 0x1;
   if (typeName!=null) c |= 0x2;
   if (indName!=null) c |= 0x4;
   if (Integer.bitCount(c)>1) {
    return new Status(-3,"(ONT.GET ...): Only one of :RELATION, :TYPE, or :INDIVIDUAL allowed
         ");
   }
   switch (c) {
   case 0:
    if (ont instanceof org.semanticweb.owlapi.model.OWLOntology && (Boolean)params.getJavaObject
         ("IMPORTS"))
     ui.println(OWLOntology.toStringPlusImports((org.semanticweb.owlapi.model.OWLOntology)ont,
          null));
    else
     ui.println(ont.toString());
    return new StatusObject<Ontology>(ont);
   case 1:
    try {
     String rel = ont.describeRelation(relName);
     ui.println(rel);
     return new StatusObject<String>(rel);
    } catch (IllegalOperationException e) {
```

```
        return new Status(−6,"(ONT.GET ... :RELATION "+relName+"): describeRelation function failed
            ",e);
       }
      case 2:
       try {
        String type = ont.describeType(typeName);
        ui.println(type);
        return new StatusObject<String>(type);
       } catch (IllegalOperationException e1) {
        return new Status(−3,"(ONT.GET ... :TYPE "+typeName+"): describeType function failed",e1);
       }
      case 4:
       try {
        String ind = ont.describeIndividual(indName);
        ui.println(ind);
        return new StatusObject<String>(ind);
       } catch (IllegalOperationException e1) {
        return new Status(−4,"(ONT.GET ... :INDIVIDUAL "+indName+"): describe function failed",e1);
       }
      default:
       return new Status(−5, "(ONT.GET ...): Internal error");
      }
   }
 }
```

A.4 Code snippets for KB.Query-if

```
/**
  * @param formula
  * @return the QueryResult as KNOWN if fact found in KB otherwise returns UNKNOWN.
  * @throws ParseException
  */
public QueryResult query(String formula) throws ParseException {
   try {
    Formula exp = SL.formula(formula);
    QueryResult result = kBase.query(exp);
    return result;
   }
   catch (Throwable e) {
    ParseException ex = new ParseException("TransientAgent.query(): malformed term '"+formula+"'",
        0);
    ex.initCause(e);
    throw ex;
   }
 }
```

A.5 Code snippets for searchOntology

```
/**
  * @param predicate
  * This function generated all possible queries form an query using
  * agent's ontology and outputs the QueryResult
  */
 public QueryResult searchOntology(Predicate predicate) {
```

```
assert predicate != null;
TransientAgent agent = ((CasaKB) myKBase).agent;
assert agent != null;

List<java.util.Set<String>> s = new java.util.ArrayList<java.util.Set<String>>();
String expr = "";
String[] identifiers = null;
switch (predicate.type) {
case BINARY_PRED:
 identifiers = new String[3];
 identifiers[0] = predicate.function.toString();
 identifiers[1] = predicate.first.toString();
 identifiers[2] = predicate.second.toString();
 expr = identifiers[0] + " " + identifiers[1] + " " + identifiers[2];
 break;
case UNARY_PRED:
 identifiers = new String[2];
 identifiers[0] = predicate.function.toString();
 identifiers[1] = predicate.first.toString();
 expr = identifiers[0] + " " + identifiers[1];
 break;
case ZEROARY_PRED:
case SIMPLE_ID:
 identifiers = new String[1];
 identifiers[0] = predicate.function.toString();
 expr = identifiers[0];
}

if (predicate.type == PredType.SIMPLE_ID) {
 OWLOntology ont = (OWLOntology) agent.getOntology();
 for (String i : identifiers) {
  try {
   if ((ont.isType(i))||(ont.isIndividual(i))) {
    java.util.Set<String> A = new HashSet<String>();
    A = recursiveParent(agent, i);
    s.add(A);
   }
  } catch (IllegalOperationException e) {
   agent.println("error", "OntologyFilter.searchOntology:", e);
  }
 }
} else {
 int arg = 0;
 OWLOntology ont = (OWLOntology) agent.getOntology();
 for (String i : identifiers) {
  try {

   if ((ont.isType(i))||(ont.isIndividual(i))) {
    java.util.Set<String> A = new HashSet<String>();
    if (arg == 0) {
     //Reverse subsumption
     A = recursiveChild(agent, i);
    } else {
     //Forward subsumption
     A = recursiveParent(agent, i);
    }
    arg++;
    s.add(A);
```

```
      }
     } catch (IllegalOperationException e) {
      agent.println("error", "OntologyFilter.searchOntology:", e);
     }
    }
   }
   if (s.size() != 0) {
    //Generate all possible queries
    java.util.Set<String> result1 = recursivePermutations(s);
    //remove original query
    result1.remove(expr);
    Iterator<String> iterator = result1.iterator();
    while (iterator.hasNext()) {
     String val = "";
     if (predicate.type == PredType.SIMPLE_ID) {
      val = (String) iterator.next();
     } else {
      val = "(" + (String) iterator.next() + ")";
     }
     try {
      QueryResult queryResult = agent.query(val);
      if (queryResult != null) {
       return queryResult;
      }
     } catch (ParseException e) {
      agent.println("error", "OntologyFilter.searchOntology:", e);
     }
    }
   }
   return null;
  }
```

A.6 Code snippets for ont.isa

```
/**
 * @param child
 * @param direct If true, return only direct ancestors (parents), otherwise return all ancestors.
 * @return The set of ancestors to Class or Individual <em>child</em>.
 * @throws IllegalOperationException
 */
 public Set<String> isa(String child) throws IllegalOperationException {
  NodeSet<OWLClass> nodeSet;
  OWLClass c = findClassInClosureBySimpleName(child);
  if (c==null) {
   if (isaStrict)
    return null;
   OWLNamedIndividual ind = findIndividualInClosureBySimpleName(child);
   if (ind==null)
    return null;
   nodeSet = instanceOf(ind);
  }
  else {
   nodeSet = isa(c);
  }
  TreeSet<String> ret = new TreeSet<String>();
```

```
  for (Node<OWLClass> node: nodeSet) {
   ret.add(node.getRepresentativeElement().getIRI().getFragment());
  }
  return ret;
 }
```

A.7 Code snippets for recursiveParent

```
 /**
  * @param agent
  * @param x
  * @return an set containing all the parents of x and x itself
  */
 public java.util.Set<String> recursiveParent(TransientAgent agent, String x) {
  java.util.Set<String> SI= new HashSet<String>();
  SI.add(x);
  java.util.Set<String> parents;
  try {
   parents = agent.getOntology().isParent(x);
   for (String p : parents) {
    //Add until top of the ontology is reached
    if (!p.equals("Thing")) {
     SI.add(p);
    }
   }
  } catch (IllegalOperationException e) {
   agent.println("error", "OntologyFilter.recursiveParent:", e);
  }
  return SI;
 }
```

A.8 Code snippets for recursiveChildren

```
/**
  * @param agent
  * @param x
  * @return an set containing all the children of x and x itself
  */
 public java.util.Set<String> recursiveChild(TransientAgent agent, String x) {
  java.util.Set<String> SI = new HashSet<String>();
  SI.add(x);
  java.util.Set<String> child;
  try {
   child = agent.getOntology().isChild(x);
   for (String c : child) {
    //Add until the bottom of ontology is reached
    if (!c.equals("Nothing")) {
     SI.add(c);
    }
   }
  } catch (IllegalOperationException e) {
   agent.println("error", "OntologyFilter.recursiveChild:", e);
  }
```

```
  return SI;
 }
```

A.9 Code snippets for recursivePermutations

```
/**
 * @param sets
 * @return The set containing all the queries for the identifiers passed
 */
 public java.util.Set<String> recursivePermutations(
    List<java.util.Set<String>> sets) {
  if (sets.size() == 1) {
    java.util.Set<String> si = sets.get(0);
    return si;
  }
  if (sets.size() == 2) {
    return getPermutations(sets.get(0), sets.get(1));
  } else {
    return getPermutations(sets.get(0),
       recursivePermutations(sets.subList(1, sets.size())));
  }
 }

 /**
 * @param a
 * @param b
 * @return an set containing all possible queries for two sets of identifiers
 */
 private java.util.Set<String> getPermutations(java.util.Set<String> a,
    java.util.Set<String> b) {
  java.util.Set<String> si = new HashSet<String>();
  for (String i : a) {
    for (String j : b) {
      String x = i + " " + j;
      si.add(x);
    }
  }
  return si;
 }
```

Appendix B

Various ontologies

B.1 Biology.owl

Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
Prefix(rdf:=<http://www.w3.org/1999/02/22−rdf−syntax−ns#>)
Prefix(xml:=<http://www.w3.org/XML/1998/namespace>)
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix(rdfs:=<http://www.w3.org/2000/01/rdf−schema#>)

Prefix(biology:=<http://casa.cpsc.ucalgary.ca/ontologies/biology.owl#>)

Ontology(<http://casa.cpsc.ucalgary.ca/ontologies/biology.owl>

Declaration(Class(biology:animal))
SubClassOf(biology:mammal biology:animal)
Declaration(Class(biology:mammal))
SubClassOf(biology:dog biology:mammal)
Declaration(Class(biology:dog))
SubClassOf(biology:goldenretriever biology:dog)
Declaration(NamedIndividual(biology:rover))
ClassAssertion(biology:goldenretriever biology:rover)
)

B.2 Predicates.owl

Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
Prefix(rdf:=<http://www.w3.org/1999/02/22−rdf−syntax−ns#>)
Prefix(xml:=<http://www.w3.org/XML/1998/namespace>)
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix(rdfs:=<http://www.w3.org/2000/01/rdf−schema#>)

Prefix(Predicates:=<http://casa.cpsc.ucalgary.ca/ontologies/Predicates.owl#>)

Ontology(<http://casa.cpsc.ucalgary.ca/ontologies/Predicates.owl>

Declaration(Class(Predicates:hasskin))
SubClassOf(Predicates:hasfur Predicates:hasskin)
Declaration(Class(Predicates:hasfur))
SubClassOf(Predicates:hashair Predicates:hasfur)
Declaration(Class(Predicates:hashair))
SubClassOf(Predicates:hastailhairs Predicates:hashair)
Declaration(Class(Predicates:hastailhairs))

)

B.3 casa.owl

Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
Prefix(rdf:=<http://www.w3.org/1999/02/22−rdf−syntax−ns#>)
Prefix(xml:=<http://www.w3.org/XML/1998/namespace>)
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix(rdfs:=<http://www.w3.org/2000/01/rdf−schema#>)
Prefix(actions:=<http://casa.cpsc.ucalgary.ca/ontologies/actions.owl#>)
Prefix(events:=<http://casa.cpsc.ucalgary.ca/ontologies/events.owl#>)
Prefix(casa:=<http://casa.cpsc.ucalgary.ca/ontologies/casa.owl#>)

Ontology(<http://casa.cpsc.ucalgary.ca/ontologies/casa.owl>
Import(<http://casa.cpsc.ucalgary.ca/ontologies/actions.owl>)
Annotation(rdfs:comment "CASA Agent Infrastructure default (base) ontology"^^xsd:string)

)

B.4 Myontology.owl

Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
Prefix(rdf:=<http://www.w3.org/1999/02/22−rdf−syntax−ns#>)
Prefix(xml:=<http://www.w3.org/XML/1998/namespace>)
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix(rdfs:=<http://www.w3.org/2000/01/rdf−schema#>)

Ontology(<Myontology.owl>
Import(<http://casa.cpsc.ucalgary.ca/ontologies/casa.owl>)
Import(<http://casa.cpsc.ucalgary.ca/ontologies/biology.owl>)
Import(<http://casa.cpsc.ucalgary.ca/ontologies/Predicates.owl>)
Annotation(rdfs:comment "CASA TransientAgent default ontology"^^xsd:string)
)

Appendix C

Test files

C.1 Ontology Filter test file

```
package casa.jade;

import casa.CASAProcess;
import casa.TransientAgent;
import casa.jade.CasaKB;
import casa.ontology.owl2.OWLOntology;
import casa.ui.BufferedAgentUI;
import casa.util.Pair;

import jade.semantics.kbase.QueryResult;

import java.text.ParseException;
import java.util.Vector;


public class OwlOntologyTiming {

 static TransientAgent agent;
 static enum predEnum {
  hasskin("hasskin", −1),
  hasfur("hasfur", 0),
  hashair("hashair", 1),
  hastailhairs("hastailhairs", 2);
  private String name;
  private int code;
  private predEnum(String name, int code) {this.name = name; this.code = code;}
  @Override public String toString() {return name;}
  public int getCode() {return code;}
 }
 private predEnum preds;
 static enum typeEnum {
  animal("animal", −1),
  mammal("mammal", 0),
  dog("dog", 1),
  goldenretriever("goldenretriever", 2);
  private String name;
  private int code;
  private typeEnum(String name, int code) {this.name = name; this.code = code;}
  @Override public String toString() {return name;}
  public int getCode() {return code;}
 }
 private typeEnum types;

 static final String separator = "\t";
 static final String quote = "\"";
 static final int n = 10;
```

```
static CasaKB cacheKb =null;
static class TableRow {
 String predicate;
 String code;
 QueryResult result;
 long[] time;
 TableRow(String pred, String code, QueryResult result, long... time) {
  predicate = pred;
  this.code = code;
  this.result = result;
  this.time = time;
 }
 @Override
 public String toString() {
  StringBuilder b = new StringBuilder("="+quote+predicate+quote+separator+"="+quote+code+
      quote+separator+result);
  for (long t: time) {
   b.append(separator).append(t);
  }
  return b.toString();
 }
}
private static Vector<TableRow> table = new Vector<TableRow>() {
 @Override
 public String toString() {
  StringBuilder b = new StringBuilder();
  for (TableRow line: table) {
   b.append(line.toString()).append("\n");
  }
  return b.toString();
 }
};


/**
 * @param args
 */
public static void main(String[] args) {



 BufferedAgentUI ui = new BufferedAgentUI();
 agent = CASAProcess.startAgent(ui);
 while(!agent.isInitialized())
 {
  try {
   Thread.sleep(500);
  } catch (InterruptedException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }
 }
 OWLOntology ontology1= (OWLOntology) ((OWLOntology)agent.getOntology()).getOntology("
     Myontology");
 agent.setOntology(ontology1);
 try {
  agent.assert_("(hasfur)");
 } catch (ParseException e) {
```

```
  // TODO Auto−generated catch block
  e.printStackTrace();
 }
 try {
  agent.assert_("(hasfur mammal)");
 } catch (ParseException e) {
  // TODO Auto−generated catch block
  e.printStackTrace();
 }
 try {
  agent.assert_("(hasfur mammal mammal)");
 } catch (ParseException e) {
  // TODO Auto−generated catch block
  e.printStackTrace();
 }



 for (predEnum i: predEnum.values()) {
  String exp = "("+i+")";
  String code = ""+i.getCode();
  record(exp,code);
 }
 for (predEnum i: predEnum.values()) {
  for (typeEnum j: typeEnum.values()) {
   String exp = "("+i+" "+j+")";
   String code = ""+i.getCode()+j.getCode();
   record(exp,code);
  }
 }
 for (predEnum i: predEnum.values()) {
  for (typeEnum j: typeEnum.values()) {
   for (typeEnum k: typeEnum.values()) {
    String exp = "("+i+" "+j+" "+k+")";
    String code = ""+i.getCode()+j.getCode()+k.getCode();
    record(exp,code);
   }
  }
 }
 System.out.println(table);

 agent.exit();
}


protected static void record(String exp, String code) {
 QueryResult result;
 try {
  result = agent.query(exp); //dry run
 long[] time = new long[n];
  for (int i=0; i<n; i++) {
   long start = System.nanoTime();
   result=agent.query(exp);
   long done = System.nanoTime();
   time[i] = (done−start)/1000;
  }
  table.add(new TableRow(exp, code, result, time));
```

```
  } catch (ParseException e) {
   e.printStackTrace();
  }
 }

}
```

C.2 Other Tests File

```
package casa.jade;

import static org.junit.Assert.*;

public class OntologyFilterTest {

 TransientAgent agent;

 /**
  * @throws java.lang.Exception
  */
 @BeforeClass
 public static void setUpBeforeClass() throws Exception {
 }

 /**
  * @throws java.lang.Exception
  */
 @AfterClass
 public static void tearDownAfterClass() throws Exception {
 }

 /**
  * @throws java.lang.Exception
  */
 @Before
 public void setUp() throws Exception {
  BufferedAgentUI ui = new BufferedAgentUI();
  agent = CASAProcess.startAgent(ui);
  while(!agent.isInitialized())
  {
   Thread.sleep(500);
  }
  OWLOntology ontology1= (OWLOntology) ((OWLOntology)agent.getOntology()).getOntology("
       Myontology");
  agent.setOntology(ontology1);


  agent.assert_("(hasfur)");
  agent.assert_("(hasfur mammal)");
  agent.assert_("(hasfur mammal mammal)");
  agent.assert_("rover");
  agent.assert_("(rover)");
  agent.assert_("(hasfur rover)");
  agent.assert_("(hasfur rover rover)");
 }
```

```
/**
 * @throws java.lang.Exception
 */
@After
public void tearDown() throws Exception {
  agent.exit();
}

@Test
public final void andTest() {
  try {
  assertNotNull(agent.query("(and (hasfur dog) (hasfur mammal))"));
  assertNull(agent.query("(and (hasfur dog) (hasskin animal))"));
  assertNull(agent.query("(and (hasskin animal) (hasfur dog))"));
  assertNull(agent.query("(and (hasskin animal) (hashair mammal))"));
  } catch (ParseException e) {
    e.printStackTrace();
  }
}

@Test
public final void orTest() {
  try {
  assertNotNull(agent.query("(or (hasfur dog) (hasfur mammal))"));
  assertNotNull(agent.query("(or (hasfur dog) (hasskin animal))"));
  assertNotNull(agent.query("(or (hasskin animal) (hasfur dog))"));
  assertNull(agent.query("(or (hasskin animal) (hashair mammal))"));
  } catch (ParseException e) {
    e.printStackTrace();
  }
}

@Test
public final void notTest() {
  try {
  assertNull(agent.query("(! (hasfur dog))"));
  assertNotNull(agent.query("(! (hasskin animal))"));
  } catch (ParseException e) {
    e.printStackTrace();
  }
}

@Test
public final void individualTest() {
  try {
  assertNotNull(agent.query("rover"));
  assertNull(agent.query("tom"));
  assertNotNull(agent.query("(rover)"));
  assertNull(agent.query("(tom)"));
  assertNotNull(agent.query("(hasskin rover)"));
  assertNull(agent.query("(hasskin tom)"));
  assertNotNull(agent.query("(hasskin rover rover)"));
  assertNull(agent.query("(hasskin rover tom)"));
  } catch (ParseException e) {
    e.printStackTrace();
  }
 }
}
```

C.3 Two argument test results

Test case number	Query	Result with filter	Result without filter
1	(hasfur mammal animal)	F	F
2	(hasfur mammal mammal)	T	T
3	(hasfur mammal dog)	T	F
4	(hasfur mammal goldenretriever)	T	F
5	(hasfur animal animal)	F	F
6	(hasfur animal mammal)	F	F
7	(hasfur animal dog)	F	F
8	(hasfur animal goldenretriever)	F	F
9	(hasfur dog animal)	F	F
10	(hasfur dog mammal)	T	F
11	(hasfur dog dog)	T	F
12	(hasfur dog goldenretriever)	T	F
13	(hasfur goldenretriever animal)	F	F
14	(hasfur goldenretriever mammal)	T	F
15	(hasfur goldenretriever dog)	T	F
16	(hasfur goldenretriever goldenretriever)	T	F
17	(hasskin mammal animal)	F	F
18	(hasskin mammal mammal)	T	F
19	(hasskin mammal dog)	T	F
20	(hasskin mammal goldenretriever)	T	F
21	(hasskin animal animal)	F	F
22	(hasskin animal mammal)	F	F
23	(hasskin animal dog)	F	F
24	(hasskin animal goldenretriever)	F	F
25	(hasskin dog animal)	F	F
26	(hasskin dog mammal)	T	F
27	(hasskin dog dog)	T	F
28	(hasskin dog goldenretriever)	T	F

29	(hasskin goldenretriever animal)	F	F
30	(hasskin goldenretriever mammal)	T	F
31	(hasskin goldenretriever dog)	T	F
32	(hasskin goldenretriever goldenretriever)	T	F
33	(hashair mammal animal)	F	F
34	(hashair mammal mammal)	F	F
35	(hashair mammal dog)	F	F
36	(hashair mammal goldenretriever)	F	F
37	(hashair animal animal)	F	F
38	(hashair animal mammal)	F	F
39	(hashair animal dog)	F	F
40	(hashair animal goldenretriever)	F	F
41	(hashair dog animal)	F	F
42	(hashair dog mammal)	F	F
43	(hashair dog dog)	F	F
44	(hashair dog goldenretriever)	F	F
45	(hashair goldenretriever animal)	F	F
46	(hashair goldenretriever mammal)	F	F
47	(hashair goldenretriever dog)	F	F
48	(hashair goldenretriever goldenretriever)	F	F
49	(hastailhairs mammal animal)	F	F
50	(hastailhairs mammal mammal)	F	F
51	(hastailhairs mammal dog)	F	F
52	(hastailhairs mammal goldenretriever)	F	F
53	(hastailhairs animal animal)	F	F
54	(hastailhairs animal mammal)	F	F
55	(hastailhairs animal dog)	F	F
56	(hastailhairs animal goldenretriever)	F	F
57	(hastailhairs dog animal)	F	F
58	(hastailhairs dog mammal)	F	F

59	(hastailhairs dog dog)	F	F
60	(hastailhairs dog goldenretriever)	F	F
61	(hastailhairs goldenretriever animal)	F	F
62	(hastailhairs goldenretriever mammal)	F	F
63	(hastailhairs goldenretriever dog)	F	F
64	(hastailhairs goldenretriever goldenretriever)	F	F

Table C.1: Results for subsumption of two-argument with `(hasfur mammal mammal)` in the KB