KNOWLEDGE ACQUISITION THROUGH NATURAL LANGUAGE CONVERSATION AND CROWDSOURCING

Luka Bradeško

Doctoral Dissertation
Jožef Stefan International Postgraduate School
Ljubljana, Slovenia

Supervisor: Doc. Dunja Mladenić, Jožef Stefan Institute, Ljubljana, Slovenia

Evaluation Board:
Dr. Michael Witbrock, Chair, IBM, New York, New York
Prof. Tomaž Erjavec, Member, Jožef Stefan Institute, Ljubljana, Slovenia
Prof. Iztok Savnik, Member, Univerza v Novi Gorici, Nova Gorica, Slovenia

MEDNARODNA PODIPLOMSKA ŠOLA JOŽEFA STEFANA
JOŽEF STEFAN INTERNATIONAL POSTGRADUATE SCHOOL

Luka Bradeško

KNOWLEDGE ACQUISITION THROUGH NATURAL LANGUAGE CONVERSATION AND CROWDSOURCING

Doctoral Dissertation

PRIDOBIVANJE STRUKTURIRANEGA ZNANJA SKOZI POGOVOR TER S POMOČJO MNOŽIČENJA

Doktorska disertacija

Supervisor: Doc. Dunja Mladenić

Ljubljana, Slovenia, September 2017

To Vanessa

Acknowledgments

I would like to express great appreciation to my PhD supervisor Prof. Dunja Mladenić for her support, guidance and advice throughout my studies. I would like to thank my coauthors, collaborators and my colleagues from Jožef Stefan Institute for many valuable discussions, insights and suggestions. I would also like to thank the members of my doctoral committee, Michael Witbrock, Tomaž Erjavec and Iztok Savnik for their valuable comments and remarks. Special thanks go to Michael Witbrock, Janez Starc and Dane Sulič for multiple years of help with the design and development of Curious Cat implementation. I thank to all of our 728 users who contributed to the experiment of this KA system, and Dr. Dave Schneider in particular, Cycorp management and members of the Cycorp technical staff (including Chris Deaton and Dr. David Baxter) more generally for licensing, for their kind assistance with APIs, KR advice and other help as we constructed and tested Curious Cat. Additionally I would like to thank Marko Grobelnik and colleagues from Artificial Intelligence Laboratory (Jožef Stefan Institute) for providing a good work and research environment, enabling me to focus on the research and implementation required to finish this thesis. Similar thanks goes to AI team at Bloomberg L.P., for encouragement and allowing me to abstain from my work duties while focusing on the thesis.

Finally I wish to thank Vanessa, my parents Agata and Bernard and my brothers Marko and Valter for their support and encouragement.

This work was supported and partially financed by European Commission under LARKC (The Large Knowledge Collider: a platform for large scale integrated reasoning and Websearch, FP7-215535), MOBIS (Personalized Mobility Services for energy efficiency and security through advanced Artificial Intelligence techniques, FP7-318452), ENVISION (EmpoweriNg European SME business model Innovation, ICT-2009-249120), OPTIMUM (Multi-source Big Data Fusion Driven Pro-activity for Intelligent Mobility, H2020-MG-636160) and personally by Dr. Michael Witbrock.

Parts of this thesis were originally published as *Curious Cat-Mobile, Context-Aware Conversational Crowdsourcing Knowledge Acquisition* in *ACM Trans. Inf. Syst.*

Abstract

Acquisition of high quality structured knowledge has been one of the longstanding goals of the Artificial Intelligence (AI) research, including for the reason that having such structured knowledge of a high quality, helps to advance other research activities from the AI field. With the recent advances in crowdsourcing, the sheer number of Internet users and the commercial availability of crowdsourcing and knowledge acquisition platforms have come a new set of tools to tackle this problem. Although numerous systems and methods for crowd-sourced knowledge acquisition had been developed and solve the problem of manpower, the issues of high financial costs, task preparation, finding the right crowd, consistency, and quality of the acquired knowledge, seem to persist.

This thesis address this deficit by formalizing and implementing an approach to lower the costs and increase the quality of acquired knowledge. The approach exploits an existing knowledge base to drive the crowd-sourced KA process, user context to communicate with the right people, and again the existing knowledge, to check for consistency of the user-provided answers. The proposed approach can be incorporated into modern natural language human-computer Interaction(HCI) interfaces doing the KA tasks in the background as part of its standard interaction (primary goal) operations, making the KA process a side-goal while not hindering the main interaction.

We also implemented the proposed approach as part of the publicly available mobile assistant application, acting as a client of proposed KA platform and supporting context acquisition algorithms.

We tested the viability of the approach in experiments through multiple years, using our platform with real users around the world, and an existing large source of common sense background knowledge. These experiments show that the approach is promising, allowing some previously not possible approach angles toward crowd-sourced knowledge acquisition and yielding a high quality of the knowledge, as a side effect of trying to act as an assistant and a companion to its users.

Povzetek

Povzetek v slovenščini naj ne bo daljši od ene strani. Translate from Englsh

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

AI	...	Artificial Intelligence
AST	...	Abstract Syntax Tree
CC	...	Curious Cat (a name of the knowledge acquisition application and platform that is a side result of this thesis)
CSK	...	Common Sense Knowledge
CYC	...	An AI system (Inference Engine and Ontology), developed by Cycorp Inc.
CycKB	...	Cyc Knowledge Base (Ontology part of Cyc system)
CycL	...	Cyc Lanugage
EBKR	...	Energy Based Knowledge Representation
FCM	...	Firebase Cloud Messaging
FOIL	...	First Order Inductive Learner
GAC	...	Generic Artificial Consciousness
GOKC	...	Goal-Oriented Knowledge Collection
GUI	...	Graphical User Interface
GWAP	...	Games With A Purpose
HCI	...	Human-Computer Interaction
HMI	...	Human to Machine Interaction
JSI	...	Jožef Stefan Institute
KA	...	Knowledge Acquisition
KB	...	Knowledge Base
KDML	...	Knowledge Database Mark-up Language
KR	...	Knowledge Representation
KU	...	Kyoto University
LSA	...	Latent Semantic Analysis
MHI	...	Machine to Human Interaction
MIT	...	Massachusetts Institute of Technology
MMHHI	...	Machine Mediated Human to Human Interaction
Mt	...	Microtheory
Mts	...	Microtheories
MPI	...	Max Planck Institute
MSR	...	Microsoft Research
NL	...	Natural Language
NLP	...	Natural Language Processing
NP	...	Noun Phrase
NTU	...	National Taiwan University
NUS	...	National University of Singapore
OIE	...	Open Information Extraction
PMI	...	Point-wise Mutual Information
POI	...	Point of Interest
POS	...	Part of Speech
POS:X	...	Abbreviations for Part of Speech tags used by POS parsers

POS:CC ... Coordinating conjunction
POS:CD ... Cardinal Number
POS:DT ... Determiner
POS:EX ... Existential there
POS:FW ... Foreign Word
POS:IN ... Preposition or subordinating conjunction
POS:JJ ... Adjective
POS:JJR ... Adjective, comparative
POS:JJS ... Adjective, superlative
POS:LS ... List item marker
POS:MD ... Modal
POS:NN ... Noun, singular or mass
POS:NNS ... Noun, plural
POS:NNP ... Proper noun, singular
POS:NNPS ... Proper noun, plural
POS:PDT ... Predeterminer
POS:POS ... Possessive ending
POS:PRP ... Personal pronoun
POS:PRP$ ... Possessive pronoun
POS:RB ... Adverb
POS:RBR ... Adverb, comparative
POS:RBS ... Adverb, superlative
POS:RP ... Particle
POS:SYM ... Symbol
POS:TO ... to
POS:UH ... Interjection
POS:VB ... Verb, base form
POS:VBD ... Verb, past tense
POS:VBG ... Verb, gerund or present participle
POS:VBN ... Verb, past participle
POS:VBP ... Verb, non-3rd person singular present
POS:VBZ ... Verb, 3rd person singular present
POS:WDT ... Wh-determiner
POS:WP ... Wh-pronoun
POS:WP$ ... Possessive wh-pronoun
POS:WRB ... Wh-adverb
PTT ... Taiwanese Bulletin Board System
SKSI ... Semantic Knowledge Source Integration
SPD ... Staypoint Detection
TUW ... The University of Waikato, New Zealand
UL ... University of Leipzig
UoM ... University of Mannheim
UoR ... University of Rochester
UW ... University of Washington

Symbols

$\in$ ... Element of. Stating $x \in S$ means that x is an element of the set S.

$\wedge$ ... logical conjunction (and). The statement $A \wedge B$ is true if both A and B are true, otherwise it is false.

$\vee$ ... logical disjunction (or). The statement $A \vee B$ is true if A or B, or both true. If both are false, the statement is false.

$\forall$ ... Universal Quantifier (for all). One of the quantifiers used to be able to convert atomic formulas into propositions. For example, $\forall x \in S : P(x)$ means that the propositional function $P(x)$ is true for every x in the set S. Or shorter. $\forall x : P(x)$, means this is true in the universal set. Given yet another example, non propositional atomic formula $x > 5$ which is neither true or false, can be converted into true/false proposition by adding a quantifier: $\forall x : x > 5$.

$\exists$ ... Existential Quantifier (there exists). One of the quantifiers, that can be used to convert atomic formulas into propositions. For example, $\exists x \in S : P(x)$ means that there exists at least one x in the set S, for which the propositional function $P(x)$ is true. Or shorter, $\exists x : P(x)$ means that there exists at least one x in the universal set, for which the propositional function is true. Given yet another, non propositional atomic formula $x > 5$ is neither true or false. But when converted to proposition by adding a quantifier it becomes either true or false in the given set: $\exists x : x > 4$.

$\Longrightarrow$ ... *Material implication*, also known as *Material conditional* or simply *implication* is a logical connective that is used to form the statements like $p \implies f$, which can be read as "if p is true, then also q is true". If p is true and q is false, then the whole statement $p \implies q$ is false.

$\Longleftrightarrow$ ... *Material Equivalence* is a bi-conditional logical connective between two logical statements. In English it can read as "Id, and only if.", or "the same as". The statement $A \iff B$ is true only if both A and B are false, or both A and B are true.

Glossary

Antecedent (predicate logic) is a first half of the hypothetical proposition. It is a p part of the implies statement (see symbol $\implies$ in Chapter Symbols for explanation. In an implication $p \implies q$, p is an antecedent.

Arity (predicate logic) is a property of predicate that defines the number of parameters or operands that predicate can operate with. For example, if a predicate P has *arity* of 2, valid statements using this predicate can only be the ones with exactly 2 operands $(P(x, y), P(a, b), ...)$. Statement $P(x)$ is in this case not a valid statement, since it uses the predicate with only one parameter.

AST (Abstract Syntax Tree) is an abstract representation of Wikipedia page as parsed from DBPedia parser. Something like DOM tree for Wikipedia instead for pure HTML

Atomic Formula (predicate logic). If the predicate P has arity n, then P followed by n constants and variables is an atomic formula. Examples: $P(a), P(x), P(x, y)D(a, x)$.

Backward Chaining or backward reasoning is the opposite of the *forward chaining*. It also uses *modus ponens*, but instead of starting from the data, it starts with a list of goals and scans all the consequents of the implication rules in the KB, check if they holds and then tries to find whether antecedents supporting these consequents hold as well. The inference engine will continue to do the inference iterations until it will find an assertion or rule which has a consequent that matches a desired goal.

Consequent (predicate logic) is a second half of the hypothetical proposition. It is a q part of the implies statement (see symbol $\implies$ in Chapter Symbols for explanation. In an implication $p \implies q$, q is a consequent or apodosis.

Constant (predicate logic) is besides *variables*, *predicates*, and *quantifiers* one of the atomic parts of the *predicate logic* sentences. For example, in a sentence $P(a, x)$, a serves as a constant. Constants are usually marked with the letters from the beginning of the alphabet. In this thesis, also predicate is a constant and all constants are written either with letters or their $Names$.

Existential Quantifier (∃). For explanation see the symbol ∃ in the chapter Symbols.

First order logic can also be called *Predicate logic*. See this term for more refined definition

Forward Chaining or forward reasoning is a repeated application of *modus ponens* inference rule. It starts with the available knowledge in the KB, then it applies the existing

rules to infer new knowledge. Then in another iteration it can use newly inferred knowledge to produce even more knowledge, and this continues until the goal is reached, or there is no new knowledge during the last iteration.

Knowledge Acquisition Bottleneck is the slow process involved with the programmatic acquisition of the knowledge to be able to use it in software. It is considered a "bottleneck" because usually other parts of the software modules involved in knowledge based application (GUI, inference engine), are relatively fast to implement.

Material Equivalence ($\iff$). For explanation see the symbol $\iff$ in the chapter Symbols.

Material implication ($\implies$). For explanation, see $\implies$ in the Chapter Symbols.

Modus Ponens is a logical inference rule stating that if an antecedent of the logical implication holds, then also the consequent holds and can be inferred. For example, a statement "if person is a programmer, then he knows how to use a computer". If we get the information that this person is a programmer, we can infer that this person knows how to use a computer.

Modus Tollens is a logical inference rule stating that if an antecedent of the logical implication holds but the consequent does not, then the negation of the antecedent can be inferred. For example, a statement "if person is a programmer, then he knows how to use a computer". If we get the information that this person cannot use a computer, and consider the previous statement, we can infer that this person is not a programmer.

OIE (Open Information Extraction is a paradigm introduced by Oren Etzioni in his TextRunner system. The main idea of this paradigm is that the knowledge acquisition system is not predetermined to extract some specific facts, patterns, etc, but is open-ended, extracting large set of relational tuples without any human input.

PMI (Point-wise Mutual Information is a measure which captures co-occurrence relationship between terms in a big corpus.

predicate is a a term used in predicate logic, representing a verb template that describes properties of objects, or relationships between multiple objects.

Predicate logic, called also *First order logic* is a formal system that uses quantification over variables. This makes this logic more expressive than the *Propositional logic.* In some limited sense, *Predicate logic* could be defined as *Propositional logic* with quantifiers.

proposition. This term is often synonym for a logical *statement*, but can also mean more abstract meaning that two different statements with the same meaning represent. In *Propositional logic*, a proposition is the smallest syntactic unit. On the other hand, in *Predicate logic*, statements/sentences are broken into *constants, variables, predicates* and *quantifiers.*

Propositional function, is an atomic function in from the *Predicate logic* which is open ended (missing quantifiers) and thus cannot count as proposition. For example, $P(x)$ is a propositional function, while $\forall x P(x)$ is a proposition.

Propositional logic, also known as *sentential* or *statement logic*, is the branch of logic that operates with entire propositions/statements/sentences to form more complicated propositions/statements/sentences, and also logical relationships and properties derived from combining or altering this statements.

Quantifier (logical). Quantifiers in *Predicate logic* convert propositional functions (open ended) into proper propositions which can be true or false. For example, $P(x)$ is a propositional function, which can get converted into proper proposition using one of the quantifiers: $\forall x P(x)$. For more info look for the terms *Universal Quantifier* and *Existential Quantifier*.

Sentential logic. See the term *Propositional logic*.

SKSI (Semantic Knowledge Source Integration) is a *Cyc* sub-system for external knowledge integration.

Statement logic. Synonym for *Propositional logic*. For description see the glossary for this term.

Upper Ontology (also top-level, foundation or core ontology) is the part of ontology (or knowledge base), which defines the core objects that serve as a main knowledge building blocks to construct the full knowledge base.

Universal Quantifier (∀). For explanation check the symbol ∀ in the chapter Symbols.

Chapter 1

Introduction

An intelligent being or machine solving any kind of a problem needs knowledge over which it can apply its intelligence when inferencing in order to be able and come up with appropriate solutions. This is especially true for any knowledge-driven AI systems which constitute a significant fraction of general AI research. For these applications, getting and formalizing the knowledge that can be trusted is crucial, but not yet straight forward to achieve on scale within accessible cost boundaries. Even in the case if it happens that future research will prove the knowledge based systems to be the dead end in the evolution of AI, solving the Knowledge Acquisition Bottleneck(Wagner 2006) will be one of the required and critical stepping stones that will need to be involved.

Such knowledge needs to be acquired by some sort of Knowledge Acquisition (KA) process, either manual, automatic or semi-automatic, converted into the appropriate representation and subsequently maintained, to keep up with the changes in the underlying domains. That this is costly is becoming increasingly obvious, with the rise of chat-bots and other conversational agents and AI assistants that became popular throughout the last few years. The most developed of these (Siri, Cortana, Google Now, Alexa, Bixby), are all backed by huge financial support from their producing companies, and the lesser-known ones are still result of 7 or more person-years of efforts by individuals(Wilcox 2011; R. Wallace 2013), or smaller companies (e.g., Josh AI[1]). Even while these systems may use statistical and machine learning (e.g. deep learning) approaches, a substantial portion of their production effort lies in knowledge acquisition, which is sometimes hidden in (hand) coded rules for request/response patterns, execution work-flow constructions and corresponding actions.

Independently of the far goal of AI research (human like intelligence), which is reflected through latest instances of AI assistants, there are countless expert systems, databases and other software solutions that need maintenance and additional semantics, and all suffer from the KA bottleneck. This is making the solutions expensive, or unable to fully utilize the idea that they implement. The bottleneck is especially obvious for the systems that make use of the real-world data that is rapidly changing through time. Some of this can be reflected through revenues of the crowdsourcing platforms (Amazon Mechanical Turk, CrowdFlower).

This dissertation deals with the KA bottleneck problem by developing a novel approach to automated knowledge acquisition, using existing knowledge, contextual crowdsourcing and natural language, to make the KA a side effect of effortless interaction of users with the computer. The thesis aims to develop, implement and evaluate the approach and do a step closer towards removing the bottleneck part from the knowledge acquisition.

The approach implementation resulted in a real world mobile assistant application

[1] www.josh.ai

(called Curious Cat), which is able to use existing knowledge to automatically infer what knowledge is missing and then ask users for the answers, check for the correctness and consistency and then place the newly acquired answer on top of the existing knowledge, to repeat the KA loop. It is using the context obtained from mobile devices to ask the right users at the right time, about the right topic. The implementation has a multi objective goal, where KA is the primary goal, and an intelligent assistant and conversational agent are secondary goals, making the KA effortless and accurate while having a conversation about concepts which have some connection to the user.

The evaluation of the approach shows that the knowledge collected this way is mostly correct and consistent with the existing KB and thus shows the approach is promising and could work also in other real-world scenarios.

1.1 Scientific Contributions

This section gives an overview of scientific and other contributions of this thesis to the knowledge acquisition approaches.

1.1.1 Novel Approach Towards Knowledge Acquisition

Traditionally KA (knowledge acquisition) approach focuses on one type of acquisition process, which can be either Labor, Interaction, Mining or Reasoning(Zang et al. 2013). In this thesis we propose a novel, previously untried approach that intervenes all aforementioned types with current user context and crowdsourcing into a coherent, collaborative and autonomous KA system. It uses existing knowledge and user context, to automatically deduce and detect missing or unconfirmed knowledge(*reasoning*) and uses this info to generate crowdsourcing tasks for the right audience at the right time(*labor*). These "KA tasks" are presented to users in natural language (NL) as part of the contextual conversation (*interaction*) and the answers are parsed (*mining*) and placed into the KB after consistency checks(*reasoning*). The approach contribution can be summed up as

1. definition of the framework for autonomous and collaborative knowledge acquisition with the help of contextual knowledge (chapter 4),
2. demonstrate and evaluate the contributions of contextual knowledge and approach in general (chapters 5 and 6).

1.1.2 Knowledge Acquisition Platform Implementation as Technical Contribution

Implementation of the KA framework as a working real-world prototype which shows the feasibility of the approach and a way to connect many independent and complex subsystems. Sensor data, natural language, inference engine, huge preexisting knowledge base (Cyc[2])(Douglas Bruce Lenat 1995), textual patterns and crowdsourcing mechanisms are connected and interlinked into a coherent interactive application (chapter 5).

1.1.3 A Shift From NL Patterns to Logical Knowledge Representation in Conversational Agents

Besides the main contributions presented above, one aspect of the approach introduces a shift in the way how conversational agents are being developed. Normally the approach

[2]Cyc is a registered trademark of Cycorp, Inc.

is to use textual patterns and corresponding textual responses, sometimes based on some variables, and thus encode the rules for conversation. As a consequence of natural language interaction, the proposed KA framework is in some sense a conversational agent which is driven by the knowledge and inference rules and uses patterns only for conversion from NL to logic. This shows promise as an alternative approach to building non scripted conversational engines (chapter 4, subsection 4.4.3).

1.2 Thesis structure

The thesis starts with a short introduction of the thesis, its main contributions and the knowledge acquisition topic it covers. Chapter 2 presents the challenges that have to be overcome or addressed by this work, and separate components required to construct the working approach.

Chapter 3 presents an overview of the related work and similar approaches from the area of knowledge acquisition.

Chapter 4 describes the general, implementation agnostic knowledge acquisition approach, one of the main contributions of the dissertation. The chapter starts with the overall architecture and then works it's way through sections, explaining the approach and addressing problems highlighted in chapter 2. Each section defines required minimal logical structures to be able to explain the approach. Definitions from the sections build on top of each-other, ending in an explanatory knowledge base, and a last section which shows through the concrete example, how the separate modules work together as a coherent KA platform.

Chapter 5 presents our real world implementation of the proposed system. It describes which tools and algorithms we used for particular components of the whole approach.

Chapter 6 presents the results of the evaluation and analysis of the implemented system and the knowledge it collected during the duration of experiments.

Chapter 7 gives main remarks and conclusions about the approach and its presented implementation. It also gives directions for future research.

Chapter 2

Background and Problem Definition

This chapter describes the challenges and components that Knowledge Acquisition system such is presented in this work (*Curious Cat*) have to address and bring together into an interconnected work-flow in order to be a coherent KA system able to satisfy the goals of general (common) Knowledge Acquisition.

Curious Cat is a KA system making use of existing knowledge, logical inference, crowd-sourcing and mobile context, to trigger natural language questions at user-appropriate moments and then incorporate the answers consistently into the existing KB. To successfully do this, there are many inter connected steps addressing a broad range of as yet not completely solved problems from multiple fields of artificial intelligence, machine learning, natural language processing and human computer interaction. Additionally to that, given that the approach uses crowd-sourcing, there is an additional complexity of technical implementation and scalability.

For more structured explanation of the approach and challenges involved, this section is grouped into sub-sections describing main challenges. Each section then references our approach to it (implementation) and also related works that gives overview of similar approaches.

Curious Cat is knowledge driven, meaning that knowledge is connecting all of the components, including the user interaction, and storing of the results into the KB (section 2.1). Its user context is obtained through a mobile sensor mining in a real world application that monitors the user's activity and location through mobile GPS and accelerometer sensors. This raw data is then corrected, clustered, classified and enriched before inserted into the KB as knowledge (section 2.2. The newly asserted context can trigger forward chaining operation of the inference engine (section 2.1) which can results in logical representation of a new question (section 2.3) or a statement that the system intends to show to the user. The aforementioned logical formula is then converted to natural language (sections 2.4) and presented to the user through a mobile application. When the user answers, his NL answer is converted back to logic (section 2.4), checked by the inference engine against the existing knowledge for consistency, and inserted as new peace of knowledge into the KB (section 2.3). After interaction like that, the system determines whether to continue the conversational path with the user or not. Newly acquired knowledge is then used to check with other users for validity (section 2.5) and is then used by the inference to produce new questions/comments/suggestions (section 2.3).

2.1 Knowledge Representation, Engineering and Inference

The proposed Knowledge Acquisition (KA) approach described in chapter 4 is completely knowledge driven. One of the main building blocks of our system is its knowledge base,

which must be based on a representation language expressive enough to support the knowledge structures required to drive the behavior of the system, and to represent the wide variety of knowledge it may gather from the users. Additionally, the inference engine needs to be able to access knowledge from the KB and needs to be powerful enough to perform inference over it.

2.1.1 Knowledge Base

The Knowledge Base (KB) is a crucial part of our KA approach (marked in purple in Figure 4.1). It dictates the expressiveness of the knowledge representation that we must use, as well as (together with the inference engine) the overall speed of the KA system. Although our approach is generally KB independent, each KB has its own specific characteristics, which need to be taken into consideration when representing and storing the acquired knowledge. As a part of this research we have considered three knowledge bases and the appropriateness of their knowledge representations.

1. The main system reported here was built based on the Cyc KB (Douglas Bruce Lenat 1995), in a form similar to Research Cyc.

2. Inspired by the Open Cyc representation we also developed our own knowledge base and inference engine, Umko, designed to be as simple as possible while still supporting several of the typical use-cases presented here. Because of its simplicity, the approach built around Umko can run in totality on a 2015-quality smart phone.

3. Additionally, to test the generality of the approach, we used similar idea with a standard RDF knowledge representation(Bradeško and Mladenić 2012).

Requirements and specifics about KB are described in more detail in section 4.2 and also in the implementation subsection 5.1.1.

2.1.2 Inference Engine

In addition to the KB, a second most crucial part of the proposed system is an inference engine (marked in red in Figure 4.1) that can reason over the knowledge stored in the KB. The inference engine is used to detect the missing knowledge and infer what to ask and when, based on the context that is asserted as a part of the knowledge. Similarly to the knowledge base, the inference engine influences the speed and complexity of the approach. Our approach was again twofold. The main experiments were based on the Cyc inference engine (see subsection 5.1.2) which is tightly linked to the Cyc KB and thus able to apply inference over the biggest existing common sense KB. Additionally, we developed the inference capabilities (forward and backward chaining) inside our custom developed inference engine - Umko, needed to test the generalize-ability of the proposed approach and to make it possible to run on embedded devices.

2.1.3 Knowledge Representation

The problem of representing knowledge in a computer is as old as the field of the AI research and has been tackled from various perspectives, but still not completely solved. The proposed approach relies on the Knowledge Representation (KR) which is powerful enough to describe the real world we live in. Additionally, it needs to cover knowledge about the knowledge itself (meta-knowledge), enabling inference over internal knowledge structures, questions and answers (see subsection 4.2.3 for approach and subsection 5.1.1 for the implementation). This inference is used to produce statements and questions and

interactions between them as part of a dialogue used to communicate with the user in the process of knowledge acquisition. We have tested two approaches, where one was fully based on the Cyc KB, and the other based on Umko, where we only created the minimal upper ontology and vocabulary to support the KA task.

2.2 Context (Information) Extraction

To be able to ask relevant questions, connected to the user or something that the user is currently doing or knows, and to ask at the right time/place, maintaining and using user context is crucial. Using context and the newly acquired knowledge to drive the KA process, is one of the main contributions of this paper.

Nowadays, the obvious information source to get to know the user is his or her mobile phone, which, with its sensors can provide an extensive and valuable source of information. However, the users need to approve using their phones sensors, and additionally, the raw measurements of sensor data need to be processed and understood to the extent that it is possible to use that information in some sort of knowledge representation. In our approach, we used two types of the context. One was provided by the users directly via answering questions about themselves (subsection 2.2.1 below and subsection 4.3.2). Another part of the context was mined from the phone sensors as introduced in subsection 2.2.2 and described in sections 4.3.1 and 5.4.2.

2.2.1 Mobile Sensor Stream Analytics (Mined Context)

Mining raw sensor measurements and extracting the meaning out of the data is a sub-problem on its own. The proposed KA approach relies on mining mobile phone GPS coordinates and accelerometer sensors to extract the places where the user stays for a while, time of the visit, duration of stay, and the path taken to come there. Additionally, we are enriching these stay points with the name and the type of location. This was done by applying a two pass stay point detect (SPD) algorithm (see related work subsection 3.5.1 and the Foursquare API. This is described in more detail in sections 4.3.1 and 5.4.2. Additionally, we use accelerometer sensors, to detect user activity (walking, running, cycling, driving, still), which is provided by Google as a part of the standard Android API. The resulting information extracted from the sensors is asserted into the KB as part of its contextual knowledge. This is depicted as an orange arrow in Figure 4.1.

2.2.2 Knowledge Acquired from the User

In addition to the context that can be mined automatically from the mobile sensors, the KA process benefits substantially if it has additional information about the user. For instance, what language she/he speaks, what are his/her interests, profession, food he/she likes, etc. This knowledge can be used by the KA rules to fine tune the questions and suggestions in order to generate more questions that the user can answer. Using context knowledge can make the questions more interesting for the user (e.g., asking the user if an already stored wireless password is still valid for a particular location that the user is currently at). In addition to specific hand crafted questions targeted at users personal information, context acquisition can use gained knowledge to ask even more questions. Wherever there is some part of knowledge in the KB that can be connected to the person, the system constructs a personal question for each user. Refer to subsection 4.3.2 for the approach explanation.

2.3 Knowledge Acquisition

The main scientific contribution of this paper is a new approach to tackling the knowledge acquisition problem. The proposed approach combines natural language crowdsourcing, usage of prior knowledge, context and newly acquired knowledge. For this, we had to address many sub-problems of knowledge driven KA as described in this section. The KA logic is all encoded as meta-knowledge in the KB and driven by inference. This is represented in red (B) and purple (A) in the architectural diagram Figure 4.1.

2.3.1 Question and Statement Formulation Logic

To be able to generate questions and other statements from a KB using an inference engine, there must first be a logic vocabulary supporting that construction. This means that the KB needs to have a meta-structure (knowledge representation) which allows it to describe its own contents and be able to address assertions and other logical formulas and variables. After this is in place, the approach needs logical rules that are used by the inference engine to produce new logical queries or statements. As part of this research, we extended the existing Cyc vocabulary with supporting meta-structures and knowledge acquisition rules. This logic is described in subsection 4.2.3 and subsection 4.4.3.

2.3.2 Answer Consistency and KB Placement

. The logical queries and statements generated by the inference engine are converted to natural language (section 4.5) and presented to the user. The users answer is converted to logic (subsection 4.4.2). Then the system checks whether the answer is coherent and consistent with the existing KB. This is crucial to avoid corrupting the KB over time as more new knowledge is added through the KA process. This check is performed with the help of the inference engine, which is triggered at the time of assertion of new knowledge and detects inconsistencies when or before an attempt to assert the answer into the KB is made. This process needs to be supported by a suitable knowledge representation, providing a suitable logical vocabulary. Our implementation addressing this problem is described in detail in subsection 5.1.3

2.3.3 Maximizing Knowledge Gain

As we are generating questions and incorporating the answers into the KB, it is obvious that some questions are more worthy of being asked than others. This details depend on the specific task of the knowledge acquisition, but regardless the task, some questions and answers are more valuable than the other. While this problem was not the main focus of this work, we addressed it partially by introducing vocabulary that allows us to control the order and priority of the questions, and also a mechanism to always show interesting and user connected questions as described in subsection 5.4.1.

2.3.4 Maximizing the chance of getting an answer

Finding the right timing, location and other context to present the question to the user. In the KA process, we want to maximize knowledge gain. A part of that is maximizing the chance of getting any answer from the user. The KA process thus become a multi-objective optimization, where we need to optimize multiple parameters to obtain as many answers as possible, while maintaining quality of the answers and keeping the user engaged at the right level (not bored and not overwhelmed). This is where the context helps the most as we can ask questions which are related to the users current situation. In particular, the

user location, user activity and time of the day helps the inference engine to decide when and what to ask. This allows the system to be aware that the user is in a restaurant and not for example in a church, which would produce totally different questions. Additionally, it enables asking "how was the food?", for example, at the right moment, when the user has already eaten in a restaurant, and not when he or she has just arrived there.

2.4 Natural Language Processing

To be able to use crowdsourcing and address the users in natural language, the KA system needs to be able to convert the logical representation of questions into their natural language representations. Similarly, when getting the users answers, these answers need to be converted back to logic. In addition, to be able to drive a coherent conversation with the user, there needs to be some logic which drives the dialog and the order of the questions and responses.

2.4.1 Logic to Natural Language Conversion

Being able to convert logic that the system uses to natural language that the user can understand is crucial part of the presented KA approach. While this is in general a hard problem and still not completely solved, it is often easy to solve simple examples of natural language generation. In our approach, we have investigated two scenarios. One is using preexisting Cyc natural language capabilities; the other is a simple scenario that is a scaled down problem using Umko. Cyc has more than 90 concepts which can be applied to create natural language generation rules(Baxter et al. 2005), and most of the predicates and concepts in the KB are annotated with the rules that instruct the system on how to convert them to natural language. For the missing NL rules and also for the new vocabulary, we extended the Cyc KB.

2.4.2 Natural Language to Logic Conversion

Due to language ambiguity, NL to logic conversion is a harder problem than the logic to NL (Schneider et al. 2015) (sections 4.4 and 5.1.3). Similarly, the problem is not completely solved yet, but to some extent, it is possible to do the simple conversion using textual patterns such us the ChatScript (subsection 3.2.7 or AIML (subsection 3.2.6). For our implementation, we used SCG system based on Cyc, which additionally to textual patterns uses inference to check whether the conversions make sense and is to some extent being able to disambiguate ambiguous conversions(Schneider et al. 2015). The idea behind this is explained in sections 4.4.2 and 5.1.3

2.4.3 Dialog Formulation Logic

Maintaining a coherent dialog while asking and answering user questions is almost as important as the language generation itself. Without it, the user quickly loses focus, is unsure about what the question refers to, etc. While this is mostly the domain of purely conversational agents such as chat-bots(Bradeško and Mladenić 2012), our approach still needs to address this to some extent. In contrast to other approaches which use text patterns to encode the dialog request and responses, our dialogs are triggered by knowledge. This means that the system actually "understands" the request and response and thus provides logical sequences in the conversation. The drawback is that there can be examples which are not covered and where the system will not be able to respond or not being able

to parse them. While this is not optimal, it is not very different from the current state of the art solutions.

2.5 Crowdsourcing

While the proposed KA approach works with even a single user, it shows its full potential when used in the crowd-sourced setting. Crowdsourcing allows the system to get better knowledge coverage including specifics that only a subset users can provide. Additionally, it allows to double-check the answers entered by the user against the answers of other users. In order for the system to use crowdsourcing, it needs to be able to address the problems that come with that, which are mostly related to different opinions, deliberately entering wrong information, trolling and handling the temporal and time dependent knowledge.

2.5.1 Knowledge Quality and Truth Control

With crowdsourcing systems, the most important and most obvious problem is quality control. Normally there needs to be a person or evaluation system that checks the quality of data returned by the crowdsourcing mechanism. For our KA purposes, we address quality control on two levels. First, the inference engine uses the existing knowledge to check whether the answers are consistent. Then, after the answers come through this first filter, we have two mechanisms that can forward the newly acquired knowledge to other users able to assess it. This is described in more detail in section 4.6.

2.5.2 Handling Different Opinions

In addition to mistakes and wrong answers, users can simply have different opinions about something. Since our KA system also behaves as an assistant and needs to be consistent towards the user, it also must handle different opinions. For this reason, we allow the users to have their own beliefs about the world, which do not affect the common knowledge base, if inconsistent with it. In this w ay if the users lie or have different opinion about something, it will only affect themselves and not the rest of the users. This capability is achieved by arranging knowledge in a tree-like structure of contexts, where the upper context is not aware of the lower ones, and the knowledge in the lower tree levels overrides the knowledge in the upper levels. In Cyc this is a kind of mechanism implemented through Micro Theories (aka micro-theories, MTs) (Johan de Kleer 2013). For the explanation of the solution refer to section 4.6.

2.5.3 Handling Temporal Knowledge and Changing Knowledge

On top of different opinions, and deliberate and accidental mistakes in the KA process, an additional problem is that some knowledge that reflects the real world must change through time. This problem needs to be tackled from two perspectives. One is that the KB and inference engine need to support the temporal dimension of knowledge. The second is that the systems need somehow to capture that the change happened and data previously valid may become invalid. For the approach using Cyc we use the fact that Cyc supports time dependent knowledge through special micro-theories where the always micro-theory is the top-most one and the time dependent ones are connected to the top one through a hierarchical micro-theory structure (see section 4.6).

2.6 Work-flow and Scalability

Besides the scientific challenges, we have described related to the proposed KA approach, in order to be able to work on a broader level, technical issues and scalability had to be addressed as well. The knowledge base and inference engine if used for common sense knowledge, tend to be quite large and resource hungry. Our experiments were concluded using three Cyc instances, running on two machines and a transcript server which is used for KB syncing on a remote EC2 Amazon instance. One main machine was running two instances each having 16GB of memory. The additional machine was used mostly for testing and evaluation. All the three Cyc instances were synced over the EC2 transcript server.

Chapter 3

Related Work

In this chapter we will give an overview of approaches and related works on broader knowledge acquisition research field, information extraction, crowdsourcing and geo-spatial context mining.

Knowledge Acquisition has been addressed from different perspectives by many researchers in Artificial Intelligence over decades, starting already in 1970 as a sub-discipline of AI research, and since then resulting in a big number of types and implementations of approaches and technologies/algorithms. The difficulty of acquiring and maintaining the knowledge was soon noticed and was Goined as *Knowledge Acquisition Bottleneck* in 1977(Feigenbaum 1977). In more recent survey of KA approaches (Zang et al. 2013), authors categorize all of the KA approaches into four main groups, regarding the source of the data and the way knowledge is acquired:

- *Labor Acquisition.* This approach uses human minds as the knowledge source. This usually involves human (expert) oncologists manually entering and encoding the knowledge.
- *Interaction Acquisition.* As in Labor Acquisition, the source of the knowledge is coming from humans, but in this case the KA is wrapped in a facilitated interaction with the system, and is sometimes implicit rather than explicit.
- *Reasoning Acquisition.* In this approach, new knowledge is automatically inferred from the existing knowledge using logical rules and machine inference.
- *Mining Acquisition.* In this approach, the knowledge is extracted from some large textual corpus or corpora.

We believe this categorization most accurately reflects the current state of machine (computer) based knowledge acquisition, and we decided to use the same classification when structuring our related work, focusing more on closely related approaches and extending where necessary. According to this classification, the work presented in this dissertation fits into a hybrid approach combining all four groups, with the main focus on interaction and reasoning. As already hinted at the beginning of chapter 2, we address the problem by combining the labor and interaction acquisition, adding unique features of using user context and existing knowledge in a combination with machine reasoning to produce practically unlimited number of potential interaction acquisition tasks. These tasks are presented as a natural language questions to multiple users (as eligible by their context), which covers aspects of crowdsourcing.

One of the most similar systems (by functionality and approach) is Goal Oriented Knowledge Collection (GOKC) (subsection 3.2.2.5). Like *Curious Cat*, *GOKC* starts with

an initial seed KB which is used to infer new questions that are presented to the user. The GOKC authors first demonstrated that without generating new types of questions, the knowledge in a domain gets saturated. Then they introduced a rule which can infer new questions through concepts which are connected by the predicates. Unlike *Curious Cat*, which checks answers for validity and has a variety of question generation rules, *GOKC* has only one rule, is fixed to a specific domain and accepts all user answers, which are then filtered only by voting.

Another related system, which is a predecessor of *Curious Cat* in some aspects, is *Cyc CURE* KA system (subsubsection 3.2.1.3 which is available as part of *Cyc*, for making the internal KA easier. While *CURE* can use inference rules to produce new questions and is able to check for consistency of the answers, it doesn't use any context, conversation or crowdsourcing. It simply lists a set of questions when *Cyc* user clicks on the concept inside Cyc's internal web-GUI.

After an initial KB, extracted from Internet textual content had been gathered, the CMU text-mining knowledge acquisition system NELL (subsection 3.3.3), started to apply a crowdsourcing approach (S D S Pedro et al. 2013), using natural language questions to validate its KB. In the same fashion, as *Curious Cat*, NELL can use newly acquired knowledge, to formulate new representations and learning tasks. There are, however, distinct differences between the approaches of NELL and Curious Cat. NELL uses information extraction to populate its KB from the web, then sends the acquired knowledge to Yahoo Answers, where the knowledge can be confirmed or rejected. By contrast, *Curious Cat* formulates its questions directly to users (and these questions can have many forms, not just facts to validate), and only then sends the new knowledge to other users for validation. Additionally, *Curious Cat* is able to use context to target specific users who have a very high chance of being able to answer a question.

On the other end of the spectrum lie conversational agents (chat-bots) with a vast number of hand-scripted NL patterns and responses. Recently, some successful chat-bots have started to employ an internal KB, in addition to just scripted requests and response (subsections 3.2.6 and 3.2.7). It is used to remember facts from the conversation for later use. In this way, the chat system is actually doing sort of targeted knowledge acquisition. There has also been an attempt to extend AIML scripts with Cyc knowledge (subsection 3.2.8) by using NL patterns to match to particular logical queries, enabling a bot to answer questions from Cyc KB. Unlike responses generated due to NL pattern matches as with chat-bots, *Curious Cat* generates NL responses based on the knowledge and context driven inference. This allows our system to pro-actively engage users in sensible conversations and is thus one of the first approaches that attempts to bridge the gap between maintaining the NL conversation of chat-bots and actually understanding its content.

Something similar (converting NL to Prolog logic, answering the question and returning the result) had been tried to some extent before(Baral et al. 2007), but just as a proof of concept example covering 4 use-cases, without doing anything else.

More recently, common sense KA quiz game with a dialog system, using also prior knowledge (*ConceptNet*) and automated question generation was implemented and tested, acquiring 150,000 unique knowledge bits from 70,000 users (Otani et al. 2016).

The rest of the related work is explained in the Table 3.1 and sub-sections that follow.

Table 3.1: Structured overview of related KA systems

System	Parent	Reference	Category	Source	Representation	Prior K.	Crowds.	Context
Cyc project (Cycorp)	/	(Douglas Bruce Lenat 1995)	Labor	K. Exp.	CycL	/	/	/
ThoughtTrasure(Signiform)	/	(Mueller 2003)	Labor	K. Exp.	LAGS	/	/	/
HowNet (Keen.)	/	(Dong et al. 2010)	Labour	K. Exp.	KDML	/	/	/
OMCS/ConceptNet (MIT)	/	(Singh et al. 2002)	Labour	Public	ConceptNet	/	✓	/
GAC/Mindpixel(McKinstry)	/	(McKinstry et al. 2008)	Labour	Public	MindPixel	/	✓	/
SKSI (Cycorp)	Cyc	(Masters et al. 2007)	Labour/Integration	K. Exp.	Structured	✓	/	/
KRAKEN (Cycorp)	Cyc	(Panton et al. 2002)	Interaction	D. Exp	CycL	✓	/	/
UIA (Cycorp)	Cyc	(Witbrock, Baxter, et al. 2003)	Interaction	D. Exp	CycL	✓	/	/
Factivore (Cycorp)	Cyc	(Witbrock, Matuszek, et al. 2005)	Interaction	D. Exp	CycL	✓	/	/
Predicate Populator (Cycorp)	Cyc	(Witbrock, Matuszek, et al. 2005)	Interaction	D. Exp	CycL	✓	/	/
CURE (Cycorp)	Cyc	(Witbrock 2010)	Interaction	D. Exp	CycL	✓	/	/
OMCommons (MIT)	OMCS	(Speer 2007)	Interaction	Public	ConceptNet	✓	✓	/
Freebase (Metaweb/Google)	/	(Bollacker et al. 2008)	Interaction	Public	RDF	/	/	/
20 Questions (MIT)	OMCS	(Speer, Krishnamurthy, et al. 2009)	Game	Public	ConceptNet	/	/	/
Verbosity (CMU)		(L. V. Ahn et al. 2006)	Game	Public	/	/	✓	/
Rapport (NTU)	ConceptNet	(Y.-l. Kuo et al. 2009)	Game	Public	ConceptNet	/	✓	/
Virtual Pet (NTU)	ConceptNet	(Y.-l. Kuo et al. 2009)	Game	Public	ConceptNet	/	✓	/
GOKC (NTU)	ConceptNet	(Y.-L. Kuo et al. 2010)	Game	Public	ConceptNet	✓	✓	/
Collabio (MS)	/	(Bernstein et al. 2010)	Game	Public	/	/	✓	/
GECKA (NUS)	/	(Cambria et al. 2015)	Game	Public	EBKR	/	✓	/
Quiz (KU)	ConceptNet	(Otani et al. 2016)	Game	Public	ConceptNet	✓	✓	/
AIML (Alice foundation)	/	(R. S. Wallace 2003)	Chat-bot	/	AIML	/	/	/
Chatscript (Brilligunderstanding)	/	(Wilcox 2011)	Chat-bot	/	ChatScript	/	/	/
CyN (Daxtron Labs)	Cyc+AIML	(Wilcox 2011)	Chat-bot	/	AIML+Cyc	✓	/	/
PCW (Cycorp)	Cyc	(Matuszek, Witbrock, et al. 2004)	Mining	Web Search	AIML+Cyc	✓	/	/
Learning Reader (NU)	Cyc	(Forbus et al. 2007)	Mining	Web	CycL	✓	/	/
NELL (CMU)	/	(Mitchell et al. 2015)	Mining	Web	Predicate l.	✓	✓	/
KnowItAll(UW)	/	(Etzioni, Popescu, et al. 2004)	Mining	Web Search	text	/	/	/
Probase (MSR)	/	(Wu et al. 2012)	Mining	Web	Proprietary	/	/	/
TextRunner (UW)	KnowItAll	(Soderland et al. 2007)	Mining	Web	text	/	/	/
ReVerb (UW)	TextRunner	(Fader et al. 2011)	Mining	Web	text	/	/	/
R2A2 (UW)	ReVerb	(Etzioni, Fader, et al. 2011)	Mining	Web	text	/	/	/
ConceptMiner (MIT)	ConceptNet	(Eslick 2006)	Mining	Web Search	ConceptNet	✓	/	/
DBPedia (UL&UoM)	Wikipedia	(Lehmann et al. 2015)	Mining	Wikipedia	RDF	/	✓	/
YAGO (MPI)	Wikipedia	(Rebele et al. 2016)	Mining	Wikipedia	RDF	✓	/	/
Cyc+Wiki (TUW)	Cyc/Wikipedia	(Medelyan et al. 2008)	Mining	Wikipedia	CycL	/	✓	/
KNEXT (MPI)		(Schubert 2002)	Mining	Penn Treebank	/	/	/	/
P. Populator+FOIL (Cyc)	Predicate Populator	(Witbrock, Matuszek, et al. 2005)	Reasoning	Induction	CycL	✓	/	/
PIP (NU)	Cyc	(Sharma et al. 2010)	Reasoning	Induction	CycL	✓	/	/
AnalogySpace (MIT)	OMCS	(Speer, Lieberman, et al. 2008)	Reasoning	Analogy	ConceptNet	✓	/	/

3.1 Labor Acquisition

This category consists of KA approaches which rely on explicit human work to collect the knowledge. A number of expert (or also untrained) ontologists or knowledge engineers is employed to codify the knowledge by hand into the given knowledge representation (formal language). Labor acquisition is the most expensive acquisition type, but it gives a high quality knowledge. It is often a crucial initial step in other KA types as well, since it can help to have some preexisting knowledge to be able to check the consistency of the newly acquired knowledge. Labor Acquisition is often present in other KA types, even if not explicitly mentioned, since it is implicitly done when defining internal workings and structures of other KA processes. While we checked other well known systems that are result of Labor Acquisition, Cyc (mentioned below) is the most comprehensive of them and was picked as a starting point and main background knowledge and implementation base for this work.

3.1.1 Cyc

The most famous and also most comprehensive and expensive knowledge acquired this way, is Cyc KB, which is part of Cyc AI system (Douglas Bruce Lenat 1995). It started in 1984 as a research project, with a premise that in order to be able to think like humans do, the computer needs to have knowledge about the world and the language like humans do, and there is no other way than to teach them, one concept at a time, by hand. Since 1994, the project continued through Cycorp Inc. company, which is still continuing the effort. Through the years Cyc Inc. employed computer scientists, knowledge engineers, philosophers, ontologists, linguists and domain experts, to codify the knowledge in the formal higher order logic language CycL (Matuszek, Cabral, et al. 2006). As of 2006, the effort of making Cyc was 900 non crowd-sourced human years which resulted in 7 million assertions connecting 500,000 terms and 17,000 predicates/relations (Zang et al. 2013), structured into consistent sub-theories (Microtheories) and connected to the Cyc Inference engine and Natural Language generation. Since the implementation of our approach is based on Cyc, we give a more detailed description of the KB and its connected systems in section 5.1 on page 85. Cyc Project is still work in progress and continues to live and expand through various research and commercial projects.

3.1.2 ThoughtTreasure

Approximately at the same time(1994) as Cyc Inc. company was formed, Eric Mueller started to work on a similar system, which was inspired by Cyc and is similar in having a combination of common sense knowledge concepts connected to their natural language presentations. The main differentiator from Cyc is, that it tries to use simpler representation compared to first-order logic as is used in Cyc. Additionally, some parts of *ThoughtTreasure* knowledge can be presented also with finite automata, grids and scripts(Mueller 1999; Mueller 2003). In 2003 the knowledge of this system consisted of 25,000 concepts and 50,000 assertions. ThoughtTreasure was not so successful as Cyc and ceased all developments in 2000 and was open-sourced on GitHub in 2015[1].

3.1.3 HowNet

Started in 1999 and is an on-line common-sense knowledge base unveiling inter-conceptual relationships and inter-attribute relationships of concepts as connoting in lexicons of the

[1] https://github.com/eriktmueller/ thoughttreasure

Chinese and their English equivalents. As of 2010 it had 115,278 concepts annotated with Chinese representation, 121,262 concepts with English representation, and 662,877 knowledge base records including other concepts and attributes (Dong et al. 2010). HowNet knowledge is stored in the form of concept relationships and attribute relationships and is formally structured in KDML (Knowledge Database Mark-up Language), consisting of concepts (called semens in KDML) and their semantic roles.

3.1.4 Open Mind Common Sense (OMCS)

A crowdsourcing knowledge acquisition project that started in 1999 at the MIT Media Lab(Singh et al. 2002). Together with initial seed and example knowledge, the system was put online with a knowledge entry interface, so the entry was crowd-sourced and anyone interested could enter and codify the knowledge. OMCS supported collecting knowledge in multiple languages. It's main difference from the systems described above (Cyc, HowNet, ThoughtTreasure) is, that it used deliberate crowdsourcing and that it's knowledge base and representation is not strictly formal logic, but rather inter-connected pieces of natural language statements. As of 2013 (Zang et al. 2013), OMCS produced second biggest KB after Cyc, consisting of English (1,040,067 statements), Chinese (356,277), Portuguese (233,514), Korean (14,955), Japanese (14,546), Dutch (5,066), etc. Initial collection was done by specifying 25 human activities, where each activity got it's own user interface for free form natural language entry and also predefined patterns like "A hammer is for _____", where participants can enter the knowledge. Although OMCS started to build KB from scratch it shares a similarity to our CC system in a sense that it is using crowd-sourcing and also natural language patterns with empty slots to fill in missing parts. OMCS was later used in many other KA approaches as a prior knowledge, similar way as we use Cyc. After a few versions, OMCS was taken from public access and merged with multiple KBs and KA approaches into an ConceptNet KB[2] (Speer, Chin, et al. 2016), which is now (in 2017) part of Linked Open Data (LOD) and maintained as open-source project.

3.1.5 GAC/MindPixel

Generic Artificial Consciousness was a bold try to make a general AI based on the premise that human thinking can be simulated with answering binary yes/no questions or decisions(McKinstry et al. 2008), where humans have a natural bias toward yes answers. With this in mind, McKinstry founded GAC (later renamed to MindPixel), where Internet crowd would answer yes/no questions for shares in the company which would commercially exploit the GAC AI. Initially, one such answer which contributed to GAC knowledge base was called MindPixel, but after a while this became name for the whole system. With this idea, McKinstry's MindPixel was one of the first crowd-sourced efforts for collaborative KB building. While MindPixel shut down after McKinstry stopped working on it, it inspired *OMCS, later OpenMind* and started the crowdsourcing and crowd based cross validation of the facts. Besides the *OMCS*, a similar YES/NO crowdsourcing system lives under the name *Weegy*[3].

3.1.6 Semantic Knowledge Source Integration (SKSI)

No matter how big the underlying knowledge base is, there will always be some missing knowledge, that exists somewhere else. While with knowledge acquisition it is possible to add these missing peaces, sometimes it makes more sense to keep this knowledge externally

[2] http://conceptnet.io/
[3] http://www.weegy.com

and only link it properly so it can be used. This is especially true for fast changing data such as stock values, weather forecast information, real-time measurements of traffic/ production line, etc. In such cases it is beneficial if one can link and address this data in the same way as it would have been included in the original KB. This is the goal of *SKS*(Masters et al. 2007) In order to connect the external source, a "wrapper" knowledge needs to be defined in *Cyc* KB, which describes which concepts and instances external source represents, and how to access them (HTTP request, SQL Query, etc.).

In *SKS* system, this "wrapper" knowledge is asserted as normal *CycL* assertions using special predicates and concepts. The descriptions are structured into three layers, where first (access layer) describes how to access the data (ie. how to connect, send queries and retrieve the content). Then the second (physical schema layer), describes how the data is structured inside the original source. The third layer (logical schema) describes in *CycL*, how the data connects to *CycKB*. Example of the logical layer is semantics on how specific columns translate into the KB. For example, table *Securities*, column *Name*, translates into instance of Cyc concept *#$Equity-Security* with the *#$nameString* linked to the value of the table cell. With this approach, it is possible to seamlessly link the existing KB with external sources, and use *Cyc* inference engine and querying mechanisms on externally connected data without even noticing it's external.

3.2 Interaction Acquisition

Similarly as with Labor KA, interaction Acquisition gets the knowledge from human minds, but in this case the acquisition is an intended side effect, while users are interacting with the software as part of some other activity/task, or as part of a motivation scheme, such as knowledge acquisition games. Besides games, the interaction could be some other user interface for solving specific tasks, or a Natural Language Conversation. This type of acquisition is most strongly correlated with the approach described in this thesis, since Curious Cat uses points (gaming), to motivate users and it interacts with user in NL, while discussing various topics (concepts). It uses the conversation to set up the context and acquire (remember) user's responses and places them properly in to the KB. Sometimes the acquired knowledge is paraphrased and presented back to user to show the 'understanding', which was first tried in OMCS (subsection 3.1.4, Singh (2002)), but there only in non-conversational way as part of the input forms.

3.2.1 Interactive User Interfaces

Interactive user interfaces are the most common representation of interaction acquisition, where the user interface is constructed in a way to help user enter the data and thus make the acquisition much faster and cheaper. Historically, these systems were developed to help the labor acquisition systems, or on top of them, after parent systems reached some sort of maturity and initial knowledge stability. This is the reason why all of these systems rely or are build on top of labor acquisition (section 3.1) or mining acquisition (section 3.3) systems.

3.2.1.1 KRAKEN

This system was a knowledge entry tool which allows domain experts to make meaningful additions to CYC knowledge base, without the training in the areas of artificial intelligence, ontology development, or knowledge representation(Panton et al. 2002). It was developed as part of DARPA's Rapid Knowledge Formation (RKF) project in 2000. As its goal was to allow knowledge entry to non-trained experts, it started to use natural language entry

and is as this, a first precursor to Curious Cat system and a seed idea for it. It consists of creators, selectors, modifiers of Cyc KB building blocks, tools for consistency checks and tools for using existing knowledge to infer new things to ask. This tool, together with it's derived solutions was later re-written and integrated into Cyc as CURE system (see below). While KRAKEN and later CURE already used Natural Language generation and parsing, and started with the idea of natural language dialogue for doing the KA, the interaction, it was missing user context (user's had to select or search the concept of interest), and also crowdsourcing aspects. Kraken was also missing rules for explicit question asking. The questions were all related to the selected concept and given as a list of natural language forms.

3.2.1.2 User Interaction Agenda (UIA)

UIA was a web based user interface for KRAKEN KA tool(Panton et al. 2002; Witbrock, Baxter, et al. 2003). It worked inside a browser and it worked as responsive web-application (in 2001) by automatically triggering refresh functionality of the browser. It consisted of a menu of tools that is organized according to the recommended steps of the KE process, text entry box (query, answer, statement), center screen for the main interaction with the current tool, and a summary with a set of colored steps needed to complete current interaction. Similarly as KRAKEN itself, this interface was later improved and integrated into main Cyc system as part of CURE tool.

3.2.1.3 Content Understanding, Review, or Entry(CURE)

As mentioned above (sections 3.2.1.1 and 3.2.1.2), *CURE*(Witbrock 2010) is an improved version of *KRAKEN* system with improved *UIA* User Interface, which is integrated directly into *Cyc*. Additional to the knowledge entry mechanisms mentioned in previous two tools, *CURE* also has a web-browser plug-in that can read current web-pages, or also free text, and annotate it with possible Cyc concepts, which are click-able and they cause the knowledge entry process for clicked concept. As already mentioned, *Curious Cat* was inspired by *CURE*, and could be considered a *CURE* improvement with more precise control over the KA process, crowdsourcing component, user context and conversational ability. In the cases when *Curious Cat* mechanisms run out of responses, our system falls back to *CURE*, showing its questions to the user, instead of immediately switching the topic (see Table 5.2 in subsection 5.4.1).

3.2.1.4 Factivore

This application was a Java Applet user interface for an extended KRAKEN system, meant for quick facts entering (Witbrock, Matuszek, et al. 2005). On the back-end it used the same mechanisms and logical templates, while in the front-end it only allowed facts entering, as opposed to UIA, which also allowed rules (which ended up as not being useful).

3.2.1.5 Predicate Populator

Predicate Populator is a similar tool as *Factivore*, which instead of only collecting instances, allows to add general knowledge about classes. For example, instead of describing facts for a specific restaurant, it can collect general knowledge that is true for all restaurants (Witbrock, Matuszek, et al. 2005). The context of the KA in this case, is given by class concept, a predicate and a web-site which is parsed into CycL concepts. These are then filtered out if they do not match argument constraints of the predicate and then shown to user for selection. As part of the validation, this tool had some problems with correctly

acquired knowledge. One of the proposed solutions (never implemented), was to start using volunteers to vote about the correctness. This is already a pre-cursor idea for crowd-sourced voting mechanisms that we used in Curious Cat.

3.2.1.6 Freebase

Freebase started in 2007(Bollacker et al. 2008) and was a large (mostly instance based) crowd-sourced graph database for structured general human knowledge. Initially it was acquired from multiple public sources, mostly Wikipedia. The initial seed was then constantly updated and corrected by the community. On the user interface side, Freebase provides an AJAX/Web based UI for humans and an HTTP/JSON based API for software access. For finding knowledge and also software based editing, it uses Metaweb Query Language (MQL). A company behind freebase was bought by Google in 2010 and incorporated into a Google Knowledge Graph. In 2016 Freebase was incorporated into the Wikidata platform and shut down by Google and is no longer maintained.

3.2.1.7 OMCommons (Open Mind Commons)

This system is an interactive interface to OMCS which can respond with a feedback to user answers and maintain dialogue (Speer 2007). This is similar approach as we do with Curious Cat and shows understanding of the knowledge users enter. The mechanisms behind is by using inference engine to make analogical inferences based on the existing knowledge and new entry. Then it generates some relevant questions and asks user to confirm them. For example, as given from the original paper, *OMCommons* asks: "A bicycle would be found on the street. Is this common sense?". This is then displayed to the user with the justification for the question: "A bicycle is similar to a car. I have been told that a car would be found on the street". Users then click on "Yes/No" buttons to confirm or reject the inferred statement. The interactive interface also allows its users to refine the knowledge entered by other users and see the ratings. Users can also explore what new inferences are result of their new contributions.

3.2.2 Games

Games are a specific sub-section of interaction acquisition, where the actual acquisition is hidden or transformed into much more enjoyable process, maximizing the entertainment of the users. This type of KA was first officially introduced by Luis von Ahn in 2006 (L. von Ahn 2006; Luis von Ahn et al. 2008) under the name 'Games with Purpose' paradigm.

3.2.2.1 20Q (20 Questions)

This is a game with intentional knowledge acquisition task which focuses to the most salient properties of concepts. The game itself is a standard 20 questions game which aims to make one player figure out the concept of discussion by asking yes/no questions and then infer from the answers what the concept could be. The only difference is that the player which is asking is a computer based on OMCS knowledge base. It generates questions in NL, and according to what a player answers, it attempts to guess the concept. To decide what questions to ask, it uses statistical classification methods (Speer, Krishnamurthy, et al. 2009), to discover the most informative attributes of concepts in OMCS KB. After the user answers all the questions, including whether the detected concept was right or not, the concept and the answers will be assigned to proper cluster and thus the characteristics of the object are learned.

3.2.2.2 Verbosity

Similarly as Q20 above, Verbosity is a spoken game for two persons randomly selected online. It was inspired by Taboo board game(Hasbro n.d.) which required players to state common sense facts without mentioning the secret concept. While having similar gameplay as aforementioned board game, Verbosity was developed with the intent to collect common sense knowledge (L. V. Ahn et al. 2006). One player (narrator), gets a secret word concept and needs to give hints about the word to the other player (guesser), who must figure out the word that is described the hints. The hints take the form of sentence templates with blanks to be filled in. For example, if the word is "CAR", the narrator could say "it has wheels." In the experiments, a total of 267 people played the game and collected 7,871 facts. While these facts were mostly a good quality and it was proven that the game can be used successfully, these facts were natural language snippets and were not incorporated into any kind of structure or formal KB.

3.2.2.3 Rapport

This is a KA game based on Chinese OMCS questions, but implemented as a Facebook game to make use of the social connections inside social network. The Game helps users to make new friends or enhance connections with their existing social network by asking and answering questions and matching the answers to other users(Y.-l. Kuo et al. 2009). This game aims to enhance the experience and community engagement and thus functionality of aforementioned *Verbosity* game, by employing simultaneous interaction between all the players versus only 1 to 1 interaction between 2 community members. For evaluation, the answers where multiple users answered the same were considered valid. This game had a similarity with Curious Cat in a sense that it employed the voting mechanism for the same answers, and the repetitive questioning of the same question to multiple users. Authors found out that the agreement between same answers of the repetitive question and voting is 80% or more after at least 2 repetitions of the same question. In 6 months, *Rapport* collected 14,001 unique statements from 1,700 users. Normalized, this is 8.2 answers per user.

3.2.2.4 Virtual Pet

This is a similar game as *Rapport* in a sense that it uses *OMCS* patterns, is in Chinese and is developed by the same authors (Y.-l. Kuo et al. 2009). Instead of Facebook platform, *Virtual Pet* uses PTT (Taiwanese bulleting board system in Chinese language). Instead of direct interaction between the users themselves, users interact with virtual pet and can ask it questions and answer it's questions. In the back-end, the questions the pet asks, are actually questions from other users. This game in 6 months collected 511,734 unique pieces of knowledge from 6,899 users. Normalized this is 74,1 answers per user. While this game attracted much more answers than *Rapport*, the quality of the answers was slightly lower. Authors argue that the reasons behind both is, that users didn't interact directly, but through the virtual pet, so they were less careful whether answers are correct or not.

3.2.2.5 Goal Oriented Knowledge Collection (GOKC)

This game builds on the findings and approach of *Virtual Pet* KA game. The main improvement is to try and actually make use of the new knowledge inside a given domain (picked by the initial seed questions), to infer new questions. With this the authors tried to fix a drawback of *Virtual Pet*, that through time, the questions and answers become saturated, and the number of new questions and answers falls exponentially through time,

with respect to the number of already collected knowledge peaces. This approach is also aligned with the CC approach, which uses existing+ context and new knowledge, to drive the questions. First part of the *GOKC* paper describes analysis of the knowledge collected by *Virtual Pet* game. The second part is a description and evaluation of GOKC KA approach, where authors did 1 week experiment to show that the approach works. During that week the system inferred created 755 new questions, out of which, 12 were reported as bad. Out of these questions 10,572 answers were collected where 9,734 were voted as good. This results in the 92,07% precision. Compared to the game without question expansion (*Virtual Pet*), which has precision of 80.58%, this is an improvement.

3.2.2.6 Collabio (Collaborative Biography)

This is also a Facebook based game, with the intention to collect user's tags. While the gathered knowledge is more a set of person's tags than knowledge, it served as an inspiration to *Rapport* and *Virtual Pet*. During the experiment, *Collabio* users tagged 3,800 persons with accurate tags with information that cannot be found otherwise(Bernstein et al. 2009; Bernstein et al. 2010).

3.2.3 GECKA (The Game Engine for Common Sense Knowledge Acquisition)

GECKA can be considered as a meta GWAP (Game with a Purpose), as it is a GWAP game developing/editing tool. It was inspired by existing GWAP games (Virtual Pet, Rapport), trying to overcome their weaknesses, naming sticky factor and un-transferability of the knowledge. This is done by allowing users to build their own games (short, or long adventure games), which they can share with friends who play. While the users are designing a game, they also generate knowledge as a side effect when defining screens, interactions between the objects and other users. Authors call these knowledge pieces *POG triples* (Prerequisite, outcome, goal). Approach was tested on 20 students from Singapore Polytechnic, and was evaluated to have the accuracy of 85.7%.

3.2.4 Quiz

Large-Scale Acquisition of Commonsense Knowledge via a Quiz Game on a Dialogue System(Otani et al. 2016) is a GWAP game developed on a smart-phone as a module inside Yahoo Voice Assist application (which is installed on more than 2.5 million devices), and interacting with the users via natural language dialogue. Over the course of 8 months, 70,000 users contributed 150,000 unique knowledge knowledge pieces, being added on top of Japanese *ConceptNet*. Regarding the game mechanics, quiz participants are given clues about a certain concept. Then they try to figure it out, but from their incorrect guesses, the game can obtain knowledge about the clues. For example, a hint "this is made of milk" is given to a player. The expected answer is "a cake", but we can learn that "cheese is made of milk" when the player answers "cheese" in response to the hint. To double check the facts, the system uses similar crowdsourcing approach as *Curious Cat*, described in subsection 4.6.1. After the evaluation, the results show that the quiz game is an effective way to acquire large amounts of knowledge.

While this game shares many similarities with *Curious Cat* (prior knowledge, natural language dialogue, automated "question" formulation, crowdsourcing), the approach is more or less random (no targeting using context), and there is no validation by inference. Also, a minor mistake in the paper, authors claim that Quiz game is the first such a system to collect large amount of knowledge using dialogue system, which is not true, since they missed our work, which was published before (Luka et al. 2016).

3.2.5 Interactive Natural Language Conversation

Natural Language Knowledge Acquisition methods are special case of Interaction Acquisition systems. While almost all of the approaches already described above (under Interactive User Interfaces and Games subsections) use natural language to some extent, the language processing used is based on relatively small amount of textual patterns, or statements which are not necessary connected into a conversation. Common denominator of these systems is that they intentionally try to acquire knowledge and then use natural language statements to do this. As a side effect and as motivation for users, sometimes consequent questions and answers give a feeling of conversation. On the other side chat-bots, start with the intention to maintain an interesting conversation with the users, and have to do knowledge acquisition only to remember facts and parts of the past conversations to be able to be smart enough, so users do not lose interest. Starting with Eliza(Weizenbaum 1966), these systems evolved, mostly directed by Turning Tests(**Turing?**), implemented as Loebner competitions, trying to pass it[4]. Through the measure of these tests(Bradeško and Mladenić 2012), among a few proprietary chat-bots, two technologies evolved (*AIML, ChatScript*) to be general enough and can be used for conversational engine (chat-bot) construction and also NL knowledge acquisition.

3.2.6 AIML(Artificial Intelligence Mark-up Language)

AIML is an XML based scripting language. It allows developers of chat-bots, to construct a predefined natural language patterns and their responses. These definitions are then fed into an AIML engine, which can match user inputs with the patterns and figure out what response to write. AIMLs syntax consists mostly of input rules (categories) with appropriate output. The pattern must cover the entire input and is case insensitive. It is possible to use a wild-card (*) which binds to one or more words. The simplest example of AIML pattern with appropriate response is presented in Table 3.2. This pattern detects user's questions like "Do you have something on the menu?" and responds with "We have everything on the menu."

Table 3.2: AIML example

```
<Category>
    <pattern> Do you have * on the menu </pattern>
    <template>
        We have everything on the menu.
    </template>
</Category>
```

AIML allows recursive calls to its own patterns, which allows for some really complicated and powerful patterns, covering many examples of input. Regarding the knowledge acquisition, AIML has an option to store parts of the textual patterns as variables and thus store information for later.

The AIML example on Table 3.3 can remember keywords following "I just ate" pattern, like "I just ate pizza". If user at some point later says "I am hungry", the bot is able to respond with "Eat another pizza". In a combination with "<that>" tag, which matches previous computer's response, AIML can be used to construct specific knowledge acquisition questions (Table 3.4). The given example is using AIML 1.0, which was later improved

[4] http://www.loebner.net

Table 3.3: AIML example of saving info to variables

```
<category>
    <pattern>I just ate *</pattern>
    <template>
        Nice choice! <set name = "food"> <star/></set>
    </template>
</category>
<category>
    <pattern>I am hungry</pattern>
    <template>
        Eat another <get name = "food"/>?
    </template>
</category>
```

with AIML2.0(R. Wallace 2013) which introduced the <*Learn*> tag, but mechanism stayed mostly the same.

Table 3.4: AIML example of remembering answers on specific questions

```
<category>
    <pattern>*</pattern>
    <that>What did you order</that>
    <template>
        Was it good? <set name = "menuItem"><star/></set>
    </template>
</category>
```

While AIML language with appropriate engine can remember specific facts, the mechanism is purely keyword based and cannot really count as structured knowledge. Additionally, since it only remembers direct facts, it would be really hard to construct an acquisition of all types of food for example. AIML based chat-bots were winning Loebner's competitions in the years from 2000 to 2004, but were later out-competed by Chatscript based bots and proprietary solutions.

3.2.7 ChatScript

ChatScript is an NLP expert system consisting of textual patterns rules. It was designed by Bruce Wilcox (Wilcox 2011) and besides patterns it has mechanisms for defining concepts, triple store for facts, own inference engine POS tagger and parser. From the measure of how close the system is to pass the Turing Test as measured by Loebner's competitions, *ChatScript* surpassed *AIML*, and is its successor, since both systems are open sourced. It was designed purposely to be simpler to use and have more powerful tools for NLP and knowledge acquisition which is integral part chat-bot systems. A simple example from AIML (Table 3.2) can be re-written in much shorter form as ChatScript rule (Table 3.5).

Similarly the example from Table 3.3 can be written as:

Similarly, example from Table 3.4 in ChatScript looks like:

Table 3.5: Simple ChatScript example

```
?: (do you have * on the menu) We have everything on the menu.
```

Table 3.6: Simple ka (remembering) example

```
s: (I just ate _*) Nice choice!
s: (I am hungry) East another _0?
```

Table 3.7: Simple ChatScript question/answer/remember example

```
t: What did you order?
        a: (_*) \$menuItem=_0 Was it good?
```

While the above examples repeats the functionality of AIML, ChatScript is more powerful and can remember facts in the shape of (subject verb object) and act on them. This is done with using *ĉreatefact* and *f̂indfact* functions.

3.2.8 CyN

CyN is an AIML interpreter implementation with additional functionality to be able to access Cyc inference engine and KB for both, storing the knowledge and also for querying(Coursey 2004). This was done by introduction of new AIML tags:

- <*cycterm*> Translates an English word/phrase into a Cyc symbol.
- <*cycsystem*> Executes a CycL statement and returns the result.
- <*cycrandom*> Executes a CycL query and returns one response at random.
- <*cycassert*> Asserts a CycL statement.
- <*cycretract*> Retracts a CycL statement.
- <*cyccondition*> Controls the flow execution in a category template.
- <*guard*> Processes a template only if the CycL expression is true

3.3 Mining Acquisition

This category of KA systems try to make use of big text corpus-es available online or otherwise on some digital media. Because the core idea of writing is to share information, there is a lot of knowledge in the texts that can be extracted and converted into a structured knowledge that can later be used by computers. Due to wast size and availability of the data on the Internet, mining is most often done on the web resources. Since most of the data format from these corpuses is text, these techniques are particularly strong in the using various NLP techniques, which are often combined with existing knowledge to correct mistakes and check consistency.

3.3.1 Populating Cyc from the Web (PCW)

Since whole idea of Cyc system is to gain enough knowledge through manual work, to be able to learn on itself after some point, the Cyc team is looking int into other means of knowledge acquisition which can automatize or speed-up the KA process. One of the approaches is by mining facts from the Internet by issuing appropriate search engine queries(Matuszek, Witbrock, et al. 2004). Because Cyc KB is really big, the first step of this approach is to select appropriate part of the kb (concepts and related queries) which are in the interest of the system. For initial experiments a set of 134 binary predicates was selected. These predicates were then used to scan the KB and find the missing knowledge, which was converted to CycL queries. These queries were then converted to NL and queried on a web search engine. The results are then converted back to CycL through NL to logic engine of Cyc. After the conversion these are converted back to NL and re-searched on the web, to check whether the results still hold, and then as the last step, the results are checked for consistency (whether they can be asserted into Cyc). From the initial 134 predicates, the system generated 348 queries, 4290 searches. It found 1016 facts, out of which 4 were rejected due to inconsistency, 566 rejected by search engine (not same results), 384 were already known to Cyc, and finally 61 new consistent facts were detected as valid. After human review, the findings were that only 32 facts were actually correct.

3.3.2 Learning Reader

Learning Reader(Forbus et al. 2007) is a prototype knowledge mining system that combines NLP, large KB (CycKB) and analogical inference into an automated knowledge extraction system that works on simplified language texts. The system uses Direct Memory Access Parsing (DMAP(Martin et al. 1986)) to parse text and convert it into CycL concepts which are then checked by the inference engine whether they can form correct CycL statements. The prototype consisted of 30,000 NDAP patterns (a quick approximation would be to imagine CyN patterns- chapter 3.2.8). After parsing and syntax checks, found CycL sentences were checked by the inference within CycKB, whether knowledge is new or not. Only new logical statements were then asserted. Additional feature of this prototype system is that it includes question answering mechanism, which can be used by evaluators to check what the system learned. This same mechanism is also used to try to generate new facts (elaborate) and also questions based on newly acquired knowledge. The experiment on 62 written stories improved the recall from 10% to 37% and kept the accuracy as 99.7% compared to original 100%. By using additional inference and conjecture based inference, the recall raised to 60% while accuracy dropped to 90.8%.

3.3.3 Never Ending Language Learner (NELL)

NELL(Mitchell et al. 2015) is a text mining KA system running 24/7 with the goal to extract knowledge, use this knowledge to improve itself and extract more knowledge. NELL was started in January 2010 and as of 2015 acquired 80 million confidence weighted new beliefs. NELL consists of many different learning tasks (for different types of knowledge), where each task also consist of the performance metrics, so the system can asses itself and check if the learning task itself is also improving through time. In 2015 it consisted of 2500 learning tasks. Some of learning task examples:

- Category Classification
- Relation Classification

- Entity Resolution
- Inference Rules among belief triples

After learning tasks, there is a *Coupling Constrains* component, which combines results of learning tasks. The potentially useful knowledge gets asserted into the KB as candidates, where the assertions are checked by knowledge integrator module which integrates the assertions into the KB, or rejects them.

After some initial initial KB had been gathered, the CMU text-mining knowledge NELL also started to apply a crowdsourcing approach(Saulo D. S. Pedro et al. 2012), using natural language questions to validate its KB. In a similar fashion as Curious Cat, NELL can use newly acquired knowledge, to formulate new representations and learning tasks. There is, however, a distinct difference between the approaches of NELL and Curious Cat. NELL uses information extraction to populate its KB from the web, then sends the acquired knowledge to Yahoo Answers, or some other Q/A site, where the knowledge can be confirmed or rejected. By contrast, Curious Cat formulates its questions directly to users (and these questions can have many forms, not just facts to validate), and only then sends the new knowledge to other users for validation. Additionally, Curious Cat is able to use context to target specific users who have a very high chance of being able to answer a question.

3.3.4 KnowItAll

KnowItAll(Etzioni, Popescu, et al. 2004) is a domain independent web fact extraction system that s specific search engine queries to find new instances of specific classes. It starts it extraction with a small seed of class names and NL patterns like "NP1 such as NP2". The classes and patterns are then used to find instances, new classes and also new extraction phrases by analyzing the results of the web search engines. For example, Googling: "Cities such as *", will return a lot of statements with instances of cities. After these cities are extracted, the names can be used for further Googling and by analyzing the phrases in which these cities appear, new patterns can be found, and so on. As part of the experiment, KnowItAll ran for 5 days and extracted over 50,000 instances of cities, states, countries, actors and films. To asses the correctness of the extractions, the system can fill-in the instances into the various patterns and check the hit-count returned by the search engine./ This then compares by the hit-count of the instance itself and uses this to asses the probability of the instance really belonging to the detected class. For example: comparing the hit-count of "Ljubljana", "Cities such as Ljubljana" and "Planets such us Ljubljana", the system can figure out that Ljubljana is most likely indeed an instance of the class city.

KnowItAll was the first of the systems that inspired an *Open Information Extraction (Open IE)* paradigm(Etzioni, Fader, et al. 2011) which resulted in many other IE systems such us TextRunner, ReVerb and R2A2. The main idea of this paradigm is to avoid hand labeled examples and domain specific verbs and nouns when approaching textual patterns which can lead to open (without specifying the targets) knowledge extraction on a web scale.

3.3.5 Probase

Probase is a probabilistic taxonomy of concepts and instances consisting of 2.7 million of concepts extracted from 1.68 billion of web pages (Wu et al. 2012). The main difference between Probase and other KBs is that Probase is probabilistic as opposed of "black and white" KB. On the other hand, even if it has much more concepts, it is sparse in the

knowledge, since it only uses *isA* relation (taxonomy). Probase was compared to other taxonomies such as WordNet, YAGO and Freebase in the sense of recall and precision of *isA* relations. Probase was found to be most comprehensive (biggest recall), while losing at precision measure against YAGO (92.8% vs 95%).

Probase was later renamed as *Microsoft Concept Graph*, and has accessible API[5] which in 2017 consists of 5,401,933 concepts, 12,551,613 instances and 87,603,947 isA relations.

3.3.6 TextRunner

TextRunner is a successor of KnowItAll system (Soderland et al. 2007) and is the first to introduce Open Information Extraction (OIE) paradigm, which's main idea is that it is open-ended and can extract information autonomously without any human intervention which would fix the system to some specific domain or set of concepts/relations. *TextRunner* was ran through over 9 million of web pages, and compared to *KnowItAll* reduced the error rate for 33% on comparable set of extractions. Throughout the experiments, *TextRunner* collected 11 million of high probability tuples and 1 million concrete facts. *TextRunner* consists of three components.

Self-Supervised Learner is a component started first which takes a small corpus of documents as an input and then outputs a classifier that can detect candidate extractions and classify them as trustworthy or not.

Single-Pass Extractor is a component that makes a single pass through the full corpus and extracts tuples for all possible relations. These tuples are then sent to the classifier trained before, which then marks them as trustworthy or not. Only trustworthy tuples are retained.

Redundancy-Based Assessor is the last step which assigns a probability to each retained tuple, based on the probabilistic model or redundancy (Downey et al. 2005).

3.3.7 ReVerb

With the experiments done with *TextRunner* and *WAE*, it became obvious that OIE systems have a lot of noise and inconsistencies in the results. For this reason two syntactical and lexical constraints were introduced in *ReVerb* OIE system(Fader et al. 2011). This helps with removing the incoherent extractions such as "recalled began" which was extracted from sentence "They recalled that Nungesser began his career as precinct leader", or uninformative extractions like "Faust, made a deal" extracted from "Faust made a deal with the devil".(Fader et al. 2011). When started, *ReVerb* first identifies relation phrases that match the constraints, then if finds appropriate pair of appropriate noun phrase arguments for each identified phrase. The resulting extractions are then given a confidence score using logistic regression classifier.

3.3.8 R2A2

R2A2 is another improvement in OIE paradigm, since previous systems assumed that relation arguments are only simple noun phrases. Analysis of *ReVerb* errors showed that 65% of errors is on the arguments side (the relation was ok). To fix this, *R2A2* system goes somehow into the direction of kb based KA systems like Curious Cat with argument constrains.(Etzioni, Fader, et al. 2011). The difference is that *R2A2* is not using hard logic and inference, but rather statistical classifier to detects class constraints (bounds) of the arguments. Compared to *ReVerb*, *R2A2* has much higher precision and recall.

[5] https://concept.research.microsoft.com

3.3.9 ConceptMiner

ConceptMiner is a KA system built by Ian Scott Eslick as part of his master thesis(Eslick 2006), with the main hypothesis that the seed knowledge collected from volunteers can be then used to bootstrap automatic knowledge acquisition. *ConceptMiner* specifically focuses on binary semantic relationships such as cause, effect, intent and time. The system relies on the prior volunteer knowledge from *ConceptNet* and tests its hypothesis with experimental extractions of knowledge around three semantic relations: desire, effect and capability.

As a first step, the system uses knowledge around predicates *DesireOf*, *EffectOf* and *CapableOf* from *ConceptNet*, to construct web-search queries. The results of these are then used to derive general patterns for aforementioned relations. For example an existing knowledge *(DesireOf "dog" "attention")*, when converted to search engine query: "dog * bark", results in patterns like:

- "My/PRP$ dog/NN loves/VBZ attention/NN ./."
- "Horseback/NN riding /VBG dog /NN attracts/VBZ attention/NN."

While not all of the patterns are of the same quality, with the sheer number of repetitions, it is possible to extract more probable ones. This then results in general patterns such as *<X>/NN loves/VBZ <Y>/NN*. These can be then used to issue a lot of search queries with various combinations of words, to extract instances of 'who desires what'. These potential instances then go into the last step (filtering). As part of this step, *ConceptMiner* removes badly formed statements, concepts not included in *ConceptNet* KB, and concepts with low PMI score (see abbreviations and glossary).

3.3.10 DBPedia

DBPedia is crowd-sourced RDF KB, extracted from Wikipedia pages and made publicly available(Lehmann et al. 2015). As of 2017 the English DBPedia contains 4.58 million knowledge pieces, out of which 4.22 million in a consistent ontology, including 1,445,000 persons, 735,000 places, 411,000 creative works (123,000 music albums, 87,000 films and 19,000 video games), 241,000 organizations (58,000 companies, 49,000 educational institutions), 251,000 species and 6,000 diseases[6]. *DBPedia* is also localized into 125 languages, so all-together it consists of 38.3 million knowledge pieces. It is also linked to YAGO categories.

Acquisition mechanism is automatic and consists of the following steps:

- Wikipedia pages are downloaded from dumps or through API and parsed into an Abstract Syntax Tree (AST)
- AST is forwarded to various extractor modules. For example, extractor module can find labels, coordinates, etc. Each of the extractor modules can convert it's part of AST into RDF triples.
- The collection of RDF statements as returned from the extractors is written into an RDF sink, supporting various format such as NTriples, etc.

[6] http://wiki.dbpedia.org/about

3.3.11 YAGO (Yet Another Great Ontology)

YAGO is an ontology built automatically from *WordNet* and *Wikipedia*(Suchanek et al. 2008). Latest version *YAGO3* is built from multiple languages and as of 2015 consist of 4,595,906 entities, 8,936,324 facts, 15,611,709 taxonomy facts and 1,398,837 labels (Demner-Fushman et al. 2015). The facts were extracted from Wikipedia category system and info boxes, using a combination of rule-based and heuristic methods, and then enriched with hierarchy (taxonomic) relations taken from WordNet. Since building YAGO is automatized, each next run of the script can use existing knowledge for type and consistency checking. This kind of type checking helps YAGO to maintain its precision at 95%(Suchanek et al. 2008). As of the summer 2017 YAGO3(Rebele et al. 2016) was released as open source project on GitHub[7].

3.3.12 KNEXT

With the premise that there is a lot of general knowledge available in texts, which lays beneath explicit assertional content, Schubert build *KNEXT* KA system(Schubert 2002) which extracts *general "possibilistic" propositions* from text. The main difference towards other KA mining systems is that before combining meanings from a phrase, the meanings are abstracted (generalized) and simplified. For example, abstraction of "a long, dak corridor" yields "a corridor". Or "a small office at the end of a long dark corridor" yields "an office". This kind of abstraction, together with weakening of relations into a possibilistic form, starts to represent presumptions about the world. The extraction follows five steps:

- Pre-process and POS tag the input
- Apply a set of ordered patterns to the POS tree recursively
- For each successfully matched subtree, abstract the interpretations using semantic rule patterns
- Collect the phrases expected to hold general "possibilistic" propositions
- Formulate the propositions and output these together with simple English representations.

From an example input statement "Blanche knew something must be causing Stanley's new, strange behavior but she never once connected it with Kitti Walker.", the output looks like this:

```
A female-individual may know a preposition
(:Q DET FEMALE-INDIVIDUAL) KNOW[V] (:Q DET PROPOS)
something may cause a behavior
(:F K SOMETHING[N]) CAUSE[V] (:Q THE BEHAVIOR[N])
a male-individual may have a behavior
(:Q DET MALE-INDIVIDUAL) HAVE[V] (:Q DET BEHAVIOR[N])
a behavior can be new
(:Q DET BEHAVIOR[N]) NEW[A])
a behavior can be strange
(:Q DET BEHAVIOR[N]) STRANGE[A])
a female-individual may connect a thing-reffered-to with a female
   -individual
(:Q DET FEMALE-INDIVIDUAL) CONNECT[V]
```

[7] https://github.com/yago-naga/yago3

(:Q DET THING–REFERRED–TO)
(:P WITH[P] (:Q DET FEMALE–INDIVIDUAL)))

The authors position their system as an addition to systems like *Cyc*, and conducted their KA experiments on Treebank corpora resulting in around 60% of propositions marked as "reasonable general claims"(Schubert and Tong 2003).

3.4 Reasoning Acquisition

Compared to other types of KA, *Reasoning Acquisition* in its essence, doesn't need any external data-sources, but it uses existing knowledge and machine inference to automatically infer additional facts from the existing knowledge. While deductive reasoning can come with new facts from the premises and rules, the reasoning that has a chance to produce higher value (non obvious) findings is inductive and analogical reasoning. Analogical reasoning can find new facts of some concept based on properties of similar concepts. On the other side, inductive reasoning can find find new probable rules based on current observations of the KB.

3.4.1 Cyc Predicate Populator + FOIL

With the initial experiments conducted from Labor Rule Acquisition done as part of *Factivore* and *Predicate Populator*(Witbrock, Matuszek, et al. 2005), it was found that getting inference rules from crowdsourcing and untrained human labour is ineffective and slow. For this reason, inductive logic inference mechanism based on FOIL (First Order Inductive Learner) approach (J.R. et al. 1997) was added to the system. Experiments were conducted on a set of 10 predicates from the KB, which generated 300 new rules. Of these rules, 7.5% were found to be correct and 35% correct with minor editing to make them well formed (assertible to Cyc). This way, rule acquisition was speed up for quite a lot, since previous experiments showed that human experts produce rules with the rate around three per hour, while with FOIL, they can review and double check for correctness around twenty rules per hour.

3.4.2 Plausible Inference Patterns (PIP)

The main idea of *PIP* system is to learn new plausible inference rules (patterns of plausible reasoning) by combining existing knowledge and reinforcement learning and thus improve the system's question answering abilities(Sharma et al. 2010). It is based on *Cyc* knowledge base, especially its predicate hierarchy represented by *#$genlPred* predicate, and *#$PredicateType* which represents second order collection of predicates that can be grouped together by some common features. For example *(#$genlPreds #$holds #$touches)*, means that each time something is holding something else, it is also touching it. The system scans the KB for rules containing predicates and tries to generate *PIPs* out of them, meaning that it tries to replace the predicates in the rules with the appropriate *PredicateType* which is linked to prior predicate. If there are more than 5 ways to generate the same PIP (from various predicate instances), then this new rule is accepted as "valid". Example of such PIP is:

$$(\#\$familyRelationsSlot(?x, ?y) \land \#\$familyRelationSlot(?y, ?z))$$
$$\Longrightarrow$$
$$\#\$personalAssociationPredicate(?y, ?z)$$

where *#$familyRelationsSlot* and *#$personalAssociationPredicate* are instances of *#$PredicateType* and thus collections of other predicates, and *?x,?y,?z* are variables representing other concepts (Sharma et al. 2010). This PIP or inference rule, tells the special inference algorithm that two predicates of type *#$familyRelationsSlot* can imply *#$personalAssociationPredicate*, when a proper combination of antecedent bindings on *?x,?y, ?z* is can be proved. Part of the *PIP* system is also an inference algorithm (FPEQ), which can make use of the above rule and come up with the possible solutions. As the first step, it constructs a graph connected to the concepts used in the Q/A query. Then, the algorithm searches for the assertions in the graph matching a set of existing PIP inference rules. As the last steps, it returns the matches (i.e. filled-in PIPs with specific predicates and concepts) which can answer the query. As an additional step, authors introduced reinforcement learning, where a small amount of user feedback (+1 or -1 voting) for final answers, can improve the PIP selection and also the level of generalizations when generating PIPs. Overall, the experiments showed that the approach increased Q/A recall for quite a lot (120% improvement), while minimally reduced the precision (94% of the baseline)(Zang et al. 2013).

3.4.3 AnalogySpace

This system is meant to work over big, but inexact knowledge bases (such as OMCS), where the need it to be able to make rough conclusions based on similarities, as opposed on hard and absolute logical facts. *AnalogySpace* does just that, by performing analogical closure by dimensionality reduction of semantic network (KB graph)(Speer, Lieberman, et al. 2008). This system tried to represent a new synthesis between a standard symbolic reasoning and statistical methods. For this, a similar technique to Latent Semantic Analysis (LSA) is being used, where strong assertions are used as opposed to weak semantics of word co-occurrences in the document. In the LSA matrix, on one axis are concepts from *OMCS*, and on the other axis a features of these concepts, which yields a sparse matrix of very high dimension. Then Singular Value Decomposition (SVD) is used on the matrix to reduce the dimensionality. This results in *principal components*, which represent the most important aspects of the knowledge. Then semantic similarity can be determined using linear operations over the resulting vectors. In a sense, this dimensionality reduction is acting as a generalization process for the kb. This way it is easier to calculate similarities between resulting concept vectors and can thus make generalizations based on these similarities, even if original concept didn't have some of the exact assertions that would enable inference engine to use it in the inference process and thus come with good answer. Results of experiments have shown that more than 70% of the resulting assertions were marked as true by human validators.

3.4.4 Cyc Wiki

Given that Wikipedia contains a lot of knowledge which is not structured in a way to be useful for inference engines directly, it makes sense to either try to structure it (as was done with *YAGO*), or link it with some existing ontology such as *CycKB*, which was done as part of this research task (Medelyan et al. 2008). Authors first filtered out of potential pool of Cyc concepts all non-common sense concepts, which ended with 83,897 of them. Then these concepts were string matched with Wikipedia concept names based on their name directly, or based on *#$nameString* predicate. For example, Cyc concept *#$Virgo-Constellation* would be matched to Wikipedia page *Virgo (Constellation)*. After this step, the mappings that have 1 to 1 relationship with Wikipedia pages and do not point to Wiki disambiguation pages, are considered properly aligned. But the mappings that have more than 1 Wikipedia result then go into the disambiguation phase. With this approach,

authors were able to get the precision of 96.2% and recall 64.0% when no disambiguation was needed, and precision 93% and recall 86.3% with the disambiguation part.

3.5 Acquisition of Geospatial Context

This sub section covers the works and approaches that can relate to our context mining/acquisition implementation. While the approaches themselves are more from the data-mining domain, as opposed to knowledge acquisition, in *Curious Cat*, the results are converted in the knowledge and asserted directly as a contextual knowledge about the users. For this reason, this related work is in its own subsection. In the Table 3.1, the approaches here are not listed, since they don't count as a knowledge acquisition, but more a custom data-mining solutions which support our proposed KA approach.

3.5.1 Extracting Places from Traces of Locations

This algorithm (we refer to it as SPD1 - Staypoint Detection), is one of the first papers/approaches that started to mine the raw GPS data into more user friendly notion of location - place.(Kang et al. 2005) The main idea of the algorithm is to be able to cluster the raw GPS locations into the particular places (home, work, specific restaurant, etc.) that user visited. This is achieved with a combination of radius(D_t) and time-based (T_t) clustering thresholds, where a cluster is a set of GPS coordinates that are within radius ($dist(loc_1) - dist(loc_n) < D_t$) during a time which is longer than a threshold ($time(loc_n) - time(loc_1) > T_t$). With this approach, it is possible to cluster locations based on how long one stays within a region, without knowing the number of clusters in front (as is necessary with more standard clustering approaches). The core of the algorithm has a benefit to be really simple and can be easily ran on the phone.

The algorithm pseudo-code is shown below (algorithm 3.1), where CL represents currently detecting cluster (stay-point), VP is detected cluster (stay-point or Visit Point) and *plocs* is temporal locations to be discarded or added to cluster later. Regarding the thresholds, T_t is Time threshold, T_d is distance threshold.

While the above algorithm robustly detects stay-points, it completely ignores paths between. Additional weak-point is that it doesn't handle well when more than one GPS coordinate wrongly jumps due to GPS accuracy. This was fixed by the improved algorithm we developed as part of *Curious Cat* system, and simultanously, the GPS accuracy part as well by the algorithm described below (subsection 3.5.2).

3.5.2 Discovery of Personal Semantic Places based on Trajectory Data Mining

This work(Lv et al. 2016)(SPD2), builds on top of the first *SPD* algorithm described in subsection 3.5.1. It improves the stay-point detection and incorporates it into the broader system which is able to map particular visits into the exact places (points of interest - POIs) and then additionally able to predict next locations of tracked users. With this functionality (especially mapping to exact points of interest), this approach shares a lot of similarities with the context mining approach that is employed within the *Curious Cat* system. The main improvement of this algorithm, is that it takes into the account the discontinuous characteristics of phone GPS signal (it gets lost inside buildings, its accuracy drops under the trees). This is done by introducing a second T_{dt} threshold, which is must be higher than T_d, and sets a tolerated distance between two consequent stay-points to be merged on the fly. Additionally, these thresholds are not fixed, but dynamically calculated based on the distribution of distances between raw GPS points.

Algorithm 3.1: Staypoint Detection Algorithm 1 (SPD1)

```
Data: raw GPS coordinates
Result: cluster representing one stay-point
if distance(CL, loc) < D_t then
    add loc to CL;
    clear plocs;
else
    if plocs.length > 1 then
        if duration(cl) > T_t then
            add CL to VP;
        end
        clear CL;
        add plocs.end to CL;
        clear plocs;
        if distance(CL,loc) < D_t then
            add loc to CL;
            clear plocs;
        else
            add loc to plocs;
        end
    else
        add loc to plocs;
    end
end
```

The simplified algorithm (removed unnecessary if/else statements) is shown below (algorithm 3.2), where CL represents currently detecting cluster (stay-point), PC is previous cluster and VP is detected cluster (stay-point or Visit Point). Regarding the thresholds, T_t is Time threshold, T_d is distance threshold and T_{dt} is tolerated distance threshold for later merging.

Similarly as *SPD1*, also *SPD2* has a problem of ignoring the GPS coordinates that are part of paths (or moves) between stay-points.

3.5.3 Applying Commonsense Reasoning to Place Identification

Clustering raw sensor data into locations, paths, activities and travel modes is often not enough. Besides knowing the location, time of arrival and duration of stay, contextual knowledge benefits from the information about the type of place and name, or even the exact ID from some database, where additional information can be looked up. This is part of the *Curious Cat* system, which was inspired by and is based on the related work of Marco Mamei(Mamei 2010). In this work, the author shows how *Cyc* knowledge base can be used to improve automatic place identification. The approach is using probabilistic ranking the list of candidates (retrieved from *Foursquare* in consideration of the common sense likelihood, which is based on the user profile, features of the day (exact time, noon, afternoon, morning, midday,...), previous visits and type of probable place. The approach was validated and given that the GPS accuracy of N95 phones (used in the experiment) is worse than that of phones in 2017, it achieved the accuracy of 75% for business type of places.

Algorithm 3.2: Staypoint Detection Algorithm 2 (SPD2)

Data: raw GPS coordinates
Result: cluster representing one stay-point

if *distance(CL, loc)* $< T_d$ **then**
  add loc to CL;
else if *duration(CL)*$>T_t$ **then**
  append CL to VP;
  CL = 0, PC = 0;
else if *interval(CL, PC)* $> T_t$ **and** *distance(CL,PC)*$< T_dt$ **then**
  CL = combine (CL, PC);
  append CL to VP;
  PC = 0;
else
  PC=CL;
  CL=0;
end

Chapter 4

Knowledge Acquisition Approach

This chapter defines the terms, formal structure and steps that form our proposed KA approach. First it introduces the general architecture and interaction loop that defines the sequence of interactions and steps involved in the process(section 4.1). In the second part, it formalizes the upper ontology and logical constructs required for the KA approach (section 4.2). After that, each of the crucial steps is described in more detail through examples and additions to the core logical structure defined earlier.

To make it easier to explain, formalize and understand the proposed approach for automated knowledge acquisition with prior knowledge, user context and conversational crowdsourcing, we will guide the explanations and formalizations through an example KA conversation depicted in tables 4.1 and 4.2.

Table 4.1: Minimal example of Curious Cat and user interaction.

Step num.	Interaction	
	Curious Cat	User1
1	Where are you? Are you at Joe's Pizza?	Yes.
2	What kind of thing is Joe's Pizza?	Restaurant.
3	Does Joe's Pizza have Wi-Fi?	Yes.
4	Is it fast enough to make Skype calls?	I don't know.
5	What's on the menu at Joe's Pizza?	Pizzas.
...Some time passes while the user eats...		
6	What did you order?	A car.
7	I've never heard of food or drink called 'car' before. Are you sure it's a type of food or drink?	No.
8	What did you order?	Pizza Deluxe
9	I've never heard of 'Pizza Deluxe' before. What kind of food or drink is it?	pizza.

When the system is conversing with the user, it uses all the knowledge gathered in prior conversations, other user's conversations and also this conversation, and is able to use it to further generate new comments and related questions. This is evident in the interaction step number 4 in Table 4.1. At the same time, or later, when some other user is in a similar context, the knowledge can be double checked with another user, as shown in Table 4.2. Based on the votes (confirmations or rejections) from the crowd (other users), the system can decide whether to believe the new knowledge in general, or only when it interacts with this particular user. This is explained in more detail in section 4.6.

Table 4.2: Minimal example of Truth checking interaction with the help of crowdsourcing.

Step num.	Interaction	
	Curious Cat	User2
10	Where are you? Are you at Joe's Pizza restaurant?	Yes.
11	Is it true that it has Pizza Deluxe on the menu?	Yes.

The resulting implementation (described at the end in chapter 5) of the described approach, named Curious Cat has a multi objective goal, KA is the primary goal, while having an intelligent assistant and a conversational agent are secondary goals. The aim is to perform knowledge acquisition effortlessly and accurately as a side effect, while having a conversation about concepts which have some connection to the user. At the same time, the approach allows the system (or the user) to follow the links in the conversation to other connected topics, covering and collecting more knowledge. For illustration see the example conversation sketch in Table 4.1, where the topic changes from a specific restaurant to a type of dish.

4.1 Architecture

The proposed KA system consists of multiple interconnected technologies and functionalities which we grouped into logical modules according to the problems they are solving (as also defined in chapter 2). This was done in order to minimize the complexity, improve the maintenance costs and allowing switching the implementations of separate sub-modules. Additionally such logical grouping increases the explain-ability and general understanding of the system.

On Figure 4.1 these modules are represented with the boxes, and their functionality groups are presented with the colors (see the figure legend). Arrows represent the interaction and work-flow order and initiation (the interaction is initiated/triggered from the origin of the arrow).

We can see that the central core of the system is the knowledge base (modules marked in purple and letter A). The knowledge base consists of *Upper Ontology* gluing everything together, *Common Sense Knowledge* to be able to "understand" user's world and check the answers for consistency, *Meta Knowledge* for enabling inference about its internal structures, *User Context KB* to hold current user context and *Knowledge Acquisition Rules* to drive the KA process from within the KB, using logical inference.

Next to the KB, is an *Inference Engine* that performs inference over the knowledge from the KB. Its modules are represented with the red color and letter B. The inference engine needs to be general enough to be able to perform over full KB, and should be capable of meta-reasoning (over the meta-knowledge and KA knowledge in the KB) about the KB's internal knowledge structures. In cases when the inference engine have some missing functionalities, some of these tasks can be supplemented by the *Procedural Support* module. In the proposed system, inference engine handles almost all of the core KA operations, which can be separated into the following modules:

- *Consistency Checking* module which can asses the user's answers and check whether they fit within the current KB knowledge.
- *KB Placement* module which decides where into the subtree of the KB the answer should be placed.

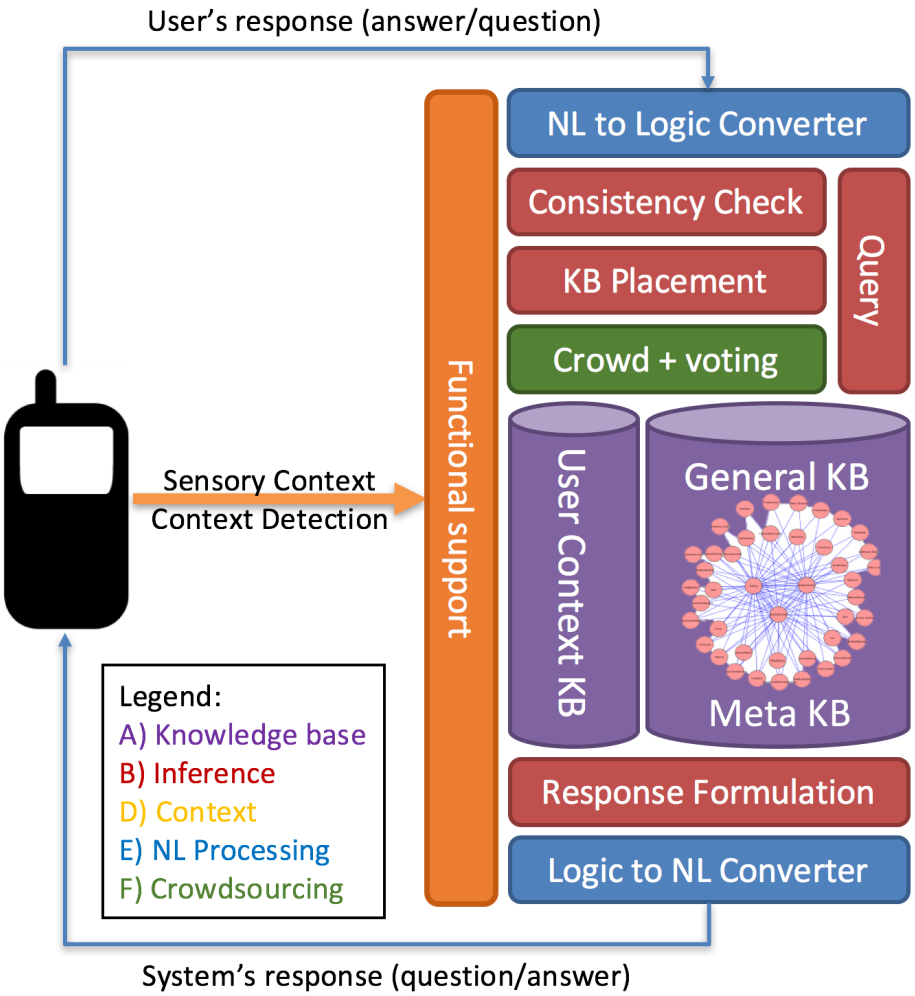

Figure 4.1: General Architecture of the KA system, with an interaction loop presented as arrows.

- *Querying* module, which employs the inference engine to answer questions that are coming from the user through NL to Logic converter.
- *Response Formulation* module, which employs the KA meta-knowledge and do inference about what to say/ask next. Results of this module are then forwarded to the Logic to NL converter and then to the user.

Tightly integrated with the knowledge base and inference engine is a *Crowdsourcing Module*, which monitors crowd (multiple user's) answers and is able to remove (or move to different contexts) the knowledge from the KB, based on its consistency among multiple users. If some piece of knowledge inside the KB is questionable, the module marks it as such and then *Response Formulation* module checks with other users whether it's true or not and should maybe be removed or only kept in the one user's part of the KB. This module is represented in Green color and letter F.

At the entry and exit point of the system work-flow, there are NLP processing modules which can convert logic into the natural language and vice versa. These modules are used for natural language communication with the users. These two modules are represented in Blue and letter E.

On the side of the Figure, there is a procedural module (depicted with Orange color

and letter D), which is a normal software module (in our implementations written in procedural programming language), which glues everything together. It contains a web-server, authentication functionalities, machine learning capabilities, connections to external services and context mining and other functions that are hard to implement using just logic and inference. This module is taking care of the interactions between submodules.

All of the modules are triggered either through the contextual triggers (also internal, like when timer detects the specific hour or time of day), or by the users. When the context changes, it causes the system to use inference engine to figure out what to do. Usually, as a consequence it results in a multiple options like questions or comments. Then it picks one and sends a request to the user. This triggering is represented with the arrows, where the blue arrows represent natural language interaction, and the orange one represents structured or procedural interaction, when the procedural module classifies or detects any useful change in the sensor data sent into the system py the part running on the mobile phone.

4.1.1 Interaction Loop

As briefly already mentioned above, besides architecture, Figure 4.1 also indicates a system/user interaction loop represented by arrows. Orange arrow (pointing directly from the phone towards the system) represents the automatic interaction or triggers that the phone (client) is sending to the system all the time. This provides one part of the user context. After the procedural part analyses the data (as described in subsection 4.3.1 and subsection 5.4.2) and enter findings into the KB as context, this often triggers the system to come up with a new question, or context related info. Example of such a trigger is, when user changes a location and the system figures out the name and type of the new place. On the other side, Blue arrows represent the Natural Language interaction which can happen as a result of automatic context (Orange arrow), or some other reason causing new knowledge appearing in the KB. New knowledge can appear as a consequence of answering a question from the same user, or some other user. This shows, how the actual knowledge (even if entered automatically through procedural component) is controlling the interaction, and explains how the system is initialized and how its main pro-activity driver is implemented. Examples of such initialization of the interaction is presented in Table 4.1 and Table 4.2. Additionally, the user can trigger a conversation at any point in time either by continuing the previous conversation or simply starting a new one.

According to the interactions described above, the proposed KA system have two options for the interaction. Human to machine (HMI), when users initiate interaction, and machine to human (MHI), when the system initiates the interaction. The specifics of both, which cannot fit in Figure 4.1 are explained in the following sub-sections 4.1.1.1 and 4.1.1.2. On top of this, the design of the system allows a novel type of interaction which combines multiple users and machine into one conversation, while presenting this to the users as a single conversation track with the machine. This becomes useful when the system doesn't have enough knowledge to be able to answer user's questions, but it has just enough to know which other users to ask (i.e. when someone is asking a question about specific place and there is no answer in the KB, *Curious Cat* can ask other users that it knows had been there). This type of the interaction can be called Machine Mediated Human to Human interaction (MMHHI). This allows the system to answer questions also when it doesn't know them, while simultaneously also store and remember the answers, either parsing them and assert them directly to KB, or leave them in NL for later Knowledge Mining analysis. The possible interaction types are also presented on Figure 4.2.

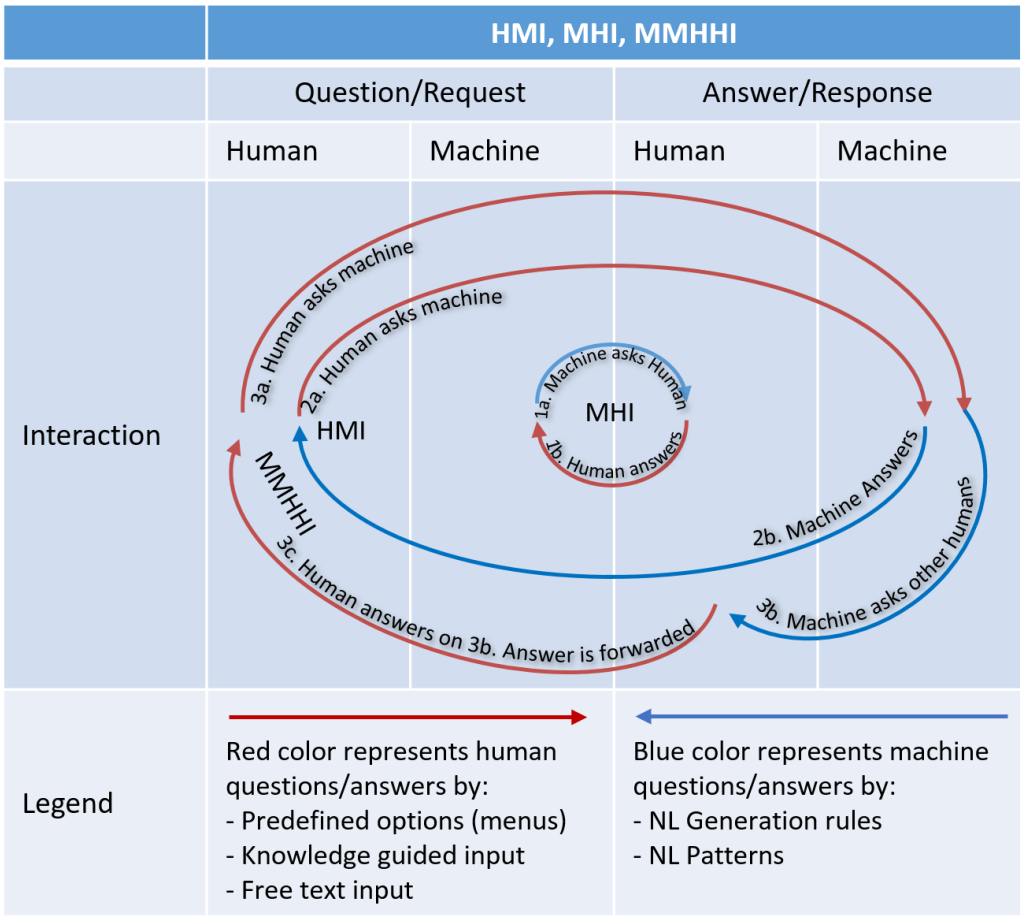

Figure 4.2: Possible interaction types between the user and Curious Cat KA System.

4.1.1.1 Machine to Human Interaction (MHI)

The most basic form of interaction between the CC system and the user, which we also use the most, is when something triggers a change in the KB and CC decides it's the time to ask or tell something. On the Figure 4.2, this is represented by the most inner loop (MHI). Example of this is when the context part of *Procedural* module classifies a new location and then asserts it into the KB. This then triggers Inference Engine which results in a new user query (same as defined in Definition 4.12).

$$ccWantsToAsk(CCUser1, (userLocation(CCUser1, CCLoc1)))$$

This query then goes through logic to NL (subsection 4.4.1) conversion, which is then presented to the user in NL like "Are you at the restaurant X?". This is represented on Figure 4.2 with a blue arrow on the inner circle, marked with 1a. The presentation is handled by the client and can be in a written form, or through the text-to-speech interface. User can then answer this question and thus close the interaction loop (blue arrow marked with 1b), possibly causing a new one with her answer.

For easier answering, the KA system can use existing KB to generate a set of possible answers at the question generation time. These can be then picked by the user instead of writing. The guidance can consist of variations of these:

- A fixed set of predefined options that user can pick from, generated from the KB.

- A set of predefined options with an additional free text field when the set of possible answers is big or infinite. In this cases the text field is connected to the KB providing auto-complete options for valid answers.

- Completely free text were user can write anything. This is essentially the same as for the HMI interaction described below (subsubsection 4.1.1.2).

The example of mediated answer guidance can be seen in Figure 5.4, where the system presented a set of possible answers while still allowing a free text which will be auto-completed with the food types that the system knows about. If the user enters something new, the system will accept that (as shown in the step 6 in Table 4.1).

The inference triggering, language rules and mechanisms for context detection are described in more detail in sections 4.4 and 4.4.3 respectively. This type of the interaction is where the users answer questions and is thus part of the main research topic of this thesis.

4.1.1.2 Human to Machine Interaction (HMI)

The second type of interaction is, when user initiates the conversation. If this is done at some point as the answer to an old question, the process is the same as described above in subsection 4.1.1.1. But users can also enter a free text, asking a question or stating something. In this case, this goes through the NL to Logic conversion (section 4.4.2 and section 5.4.2) which tries to convert the text into logical query or assertion. The complexity of converting NL into logic is a lot higher than in the opposite direction, since the language is not as exact. In CC implementation (section 5.4.2), we handled this to some extent by using SCG system (Schneider et al. 2015), where the text is matched to NL patterns which are linked to appropriate logical structures. This would be used for example, when user, instead of simple "Pizza Deluxe" (step 12 in Table 4.1, would say "They sell pizzas", or something even more complex (see section 4.4.2). After the text is converted into logic, inference engine can use it to query it against the KB, and show the answers back to user, again converted into NL through *Logic to NL Converter* module. This type of interaction is depicted on Figure 4.2 by the middle arrow circle started by humans (arrow 2a), where machine provides the response (arrow 2b).

4.1.1.3 Machine Mediated Human to Human Interaction (MMHHI)

Both of the interactions described in sections 4.1.1.1 and 4.1.1.2 presupposes that the recipient of the query, knows how to answer it, or respond otherwise. In the cases when, let's say machine doesn't have any answer (The NL question gets converted into the logical query, which doesn't retrieve any answers from within the KB). It could respond with "I don't know", which is a valid response. While this allows for the conversation to continue, it doesn't help the user to get the answer, also does not benefit to knowledge acquisition. The only thing the system can learn from this, is that user is interested in the object of the question. This doesn't have to end there though, since CC has access to other users and knowledge about their past and current contexts. Based on the topic of interest from the user query, the system can easily find users which might know the answer (inferred from their past whereabouts, answers, etc.). Once such an user is detected, the original question can simply be forwarded to him, as it would be asked by CC itself. Once he, or one of the users answers, CC can forward it to the original user. On top of that, CC can parse the answer the same way as described above for HMI and MHI (sections 4.1.1.1, 4.1.1.2), and remember it, placing it into the KB. In the cases when the language of the answer is too complex, it can be stored in its original format, for later text-mining approach which can lead to learning of new-patterns as well as the knowledge hiding in the answers. On top of

this, CC can also remember the question itself, and place it on specific type of concepts, as an important question to ask. On Figure 4.2, MMHHI approach is depicted by the outer circle of arrows, where 3a is original user's question, which is forwarded to other users when CC doesn't know how to answer (3b). After one or more of the users answer, the answer is forwarded back to the original user (3c).

4.2 Knowledge Base

As visible in Figure 4.1, Knowledge Base is the central part of the proposed KA system. Internally KB has three components. The main part, which should in any real implementation of the system also be the biggest, is the common-sense knowledge, and its upper ontology over which we operate. This part of the system contributes the most to the ability to check the answers for consistency. The more knowledge already exists, the easier becomes to assess the answers, come up with new questions and also propose possible answers in the guided interaction(subsection 4.2.2).

The second part is the user Context KB, which stores the contextual knowledge about the user. This covers the knowledge that the user has provided about himself (subsection 4.3.2) and the knowledge obtained by mining raw mobile sensors (subsection 4.3.1). On Figure 4.1, This part of the KB is represented as the left-most KB, sitting between the main KB and the *Procedural Module*.The sensor based context allows the system to pro-actively target the right users at the right time and thus improve the efficiency and accuracy and also stickiness of the KA process.

The third KB part, is the meta-knowledge and KA rules that drive the dialog and knowledge acquisition process (section 4.2.3). Although in our implementation we used Cyc KB (subsection 5.1.1), the approach is not fixed to any particular knowledge base. But the KB needs needs to be expressive enough to be able to cover the intended knowledge acquisition tasks and meta-knowledge needed for the system's internal workings.

Because the full Curious Cat system including the KB is too big and complex to be fully explained here (the KA Meta Knowledge alone consists of 12,353 assertions and rules), we will focus on the fundamentals of the idea and approach, and define the simplest possible logic to explain the workings through the examples given in the Table 4.1 and Table 4.2.

The logic examples are given in formal higher order predicate logic, where additional to variables given as predicate arguments (x, y, z), we sometimes use rules, where predicate (i.e. P) letter is also treated as variable. In these cases it is obvious that the predicate is a variable as well, because it is used in a inference rule (implication $\Longrightarrow$ or equivalence $\Longleftrightarrow$) and marked with one of the quantifiers ($\forall$ or $\exists$). When the variables are prefixed with the \$ sign, they are treated as a concept and do not represent an unbound variable, but a concept representing the variable which can appear also in the rule consequents. This is used to be able to represent the logical queries as a consequence of inference. When we have exact definitions of the predicates and constants, we try to name them accordingly, where the predicate names start with a small letter (*predicate*) and the rest of the concepts with a capital letter (*Concept*). At this point it is worth noting that while our logical definitions and formalization are strongly influenced by Cyc(Douglas Bruce Lenat 1995), and while the approach is based on the Cyc upper ontology, the approach is general and not bound to any particular implementation, and our notation below reflects but is not tightly bound to that of OpenCyc[1].

[1]The notation here follows closely practices used in Cyc. For more details, readers can refer to (Douglas Bruce Lenat 1995) and (Matuszek, Cabral, et al. 2006)) and the references it contains.

4.2.1 Upper Ontology

First we introduce the vocabulary or terms (constants/concepts) that will allow us to construct the upper ontology which is the glue of any knowledge base that can be used for machine inference:

$$S_{Constants} = \{Something, Class, Number, Predicate, subclass, is, arity, argClass, argIs\} \tag{4.1}$$

In the standard *predicate logic*, $P(x)$ notation tells us that whatever the x stands for, it has the property defined by the predicate P. For example, the following propositional function:

$$Person(x) \tag{4.2}$$

is stating that something (x) is a person, or more precisely, x is an instance of a class $Person$. In order to be able to construct logical statements ranging over classes and their instances in a more controlled and transparent way, we use a constant is, and define it as a predicate denoting that something is an instance of some class.

Definition 4.1 (predicate "is"). Predicate $is(x, y)$ denotes that x is an instance of y. For example, stating $is(John, Human)$ defines John as one instance of the class of humans.

Now, to make things clearer and more precise, instead of writing instance relation through custom predicate $Person(x)$, we can use more precise syntax, which allow us to specify what is instance of which class: $is(x, Person)$.

As visible in the Equation 4.2, in predicate logic, predicates are defined only with usage. Everything that we write as a predicate in a similar formula is then defined as a predicate. This is not a desired behavior in our KA approach, since the knowledge will be coming from the users with various backgrounds and without any idea of predicate logic. For this reason we need more control of what can be used where, if we want to be able to check for consistencies and have control of our KB with the inference engine. For this reason, we enforce a constraint (constraint rule).

Definition 4.2 (Predicate Constraint). Everything that we want to use in the KB as a predicate, must first be defined as an instance of the $Predicate$ class: $\forall x \in S_{Predicates} : is(x, Predicate)$. From the other side, set of predicates can be defined as $S_{Predicates} = \{x : is(x, Predicate)\}$. This constraint can be inserted into our KB as a *material implication* rule, which needs to be true at all times, to serve as a constraint:

$$\forall P \forall x_{1...n}(P(x_1...x_n) \implies is(P, Predicate)) \tag{4.3}$$

Now, careful reader might notice, that we actually cannot use is as a predicate, since nowhere in our KB is stated that this is actually a predicate. To fix this error and make our KB consistent with its constraints, we need to add an assertion defining what term is stands for:

$$is(is, Predicate) \tag{4.4}$$

At the time the above assertion (Assertion 4.4) is asserted into the KB, it also becomes valid assertion, since it complies with the constraint defined in Definition 4.2 and thus our Constraint Rule 4.3 is true. After this assertion is in the KB, ir can be used as a predicate because it is an instance of the term $Predicate$ and complies with our constraints.

At this point we can define(assert) the rest of our predicates:

$$is(subclass, Predicate) \wedge is(arity, Predicate) \wedge is(argClass, Predicate) \\ \wedge is(argIs, Predicate) \tag{4.5}$$

And also the rest of our terms, which we define as instances of term $Class$.

$$is(Class, Class) \wedge is(Predicate, Class) \\ \wedge is(Number, Class) \tag{4.6}$$

In Predicate logic, predicates have a property called arity, which defines number of arguments that the predicate can have. For example, if predicate P has arity of 1, then it can only take one operand (variable or term). In this case only $P(x)$ or $P(a)$ are valid statements, and $P(x, y)$ is not. In our KB, arity is defined using *arity* predicate, which itself was defined in Assertion 4.5.

Definition 4.3 (predicate "*arity*"). Predicate $arity(x, y)$ denotes that predicate x has arity of y.

Similarly, as with the constraint that all predicates need to be defined as such (Definition 4.2), we gain more control over KB and make things easier for the inference engine and KA approach, if we limit the assertions, to "obey" the predicate arities. For this reason our KB has additional constraint.

Definition 4.4 (Arity Constraint). All assertions in CC KB are valid only, when the predicates used in the assertion have the same number of arguments as defined with their *arity* assertions:

$$\forall P \forall n \forall x_{1...n} : (P(x_1, ..., x_n) \implies arity(P, n)) \tag{4.7}$$

After we add the above rule (Constraint Rule 4.7), our KB is not consistent anymore, since all the $is(x, y)$ assertions (Assertions 4.4, 4.5) violate the constraint. We fix this by adding the following assertion:

$$arity(arity, 2) \wedge arity(is, 2) \tag{4.8}$$

This makes the KB consistent again, because we defined all the arities of predicates *arity* and *is*, which we have used so far in our KB, as well as defined them with $is(x, Predicate)$ assertions. We can now continue with defining the arities of the rest of the predicates:

$$arity(subclass, 2) \wedge arity(argClass, 3) \wedge arity(argIs, 3) \tag{4.9}$$

We can see now that the arity of *is* predicate is defined as 2 (same as for *subclass* and *arity*, which can be used to define arity of itself), and can confirm that all the logical formulas in the definitions up to now are correct.

To be able to describe the world in more detail, we define the *subclass* predicate, which handles the hierarchy relations between multiple classes (unlike *is*, which handles relationships between classes and their instances).

Definition 4.5 (predicate "*subclass*"). Predicate $sublcass(x, y)$ denotes that x is a subclass of y. For example, asserting $subclass(Dog, Animal)$, is specifying all dogs, to be a sub-class of animals, and $subclass(Terrier, Dog)$ is specifying that all terriers are sub-class of dogs. At this point we might notice that while it is logical to us that terriers are also a sub-class of animals, there is no way for the machine inference to figure that out. For this reason we need to introduce a "subclass transitivity" inference rule:

$$\forall x \forall y \forall z : ((subclass(x, y) \wedge subclass(y, z)) \implies subclass(x, z)) \tag{4.10}$$

The rule above is basically saying that if a first thing is a sub-class of a the second thing, and then the second thing is a subclass of the third thing, then the third thing is a sub-class of the first thing as well.

Because we want to be able to prevent our system from acquiring incorrect knowledge, we need to limit the domains and ranges of the predicates (arguments). This could be done by adding a specific constraint rules (material implication rule without the power to make new assertions). For example, for both *subclass* arguments, to only allow instances of a $Class$, we could assert:

$$\forall x_1 \forall x_2 : (subclass(x_1, x_2) \implies (is(x_1, Class) \wedge is(x_2, Class)) \tag{4.11}$$

Because the rule (Rule 4.11 is only true if the right part (the consequent) is true, or the left part (the antecedent) is false, its inclusion in the KB (as with other constraint rules) forces the KB to not allow the arguments of subclass to be anything else than an instance of a class $Class$. It would be hard to construct a large KB, by writing the rule like this for each of the thousands potential predicates. To make this easier, following Cyc practice(Douglas Bruce Lenat 1995), we will introduce $argIs$ predicate (Definition 4.5). To make this definition more understandable, let's first expand the example (Rule 4.11 above:

$$\begin{gathered} \forall x_1 \forall x_2 : (subclass(x_1, x_2) \implies (is(x_1, Class) \wedge is(x_2, Class)) \\ \iff \\ argIs(subclass, 1, Class) \wedge argIs(subclass, 2, Class)) \end{gathered} \tag{4.12}$$

This rule above (Rule 4.12 states, that the constraint rule (Rule 4.11 can be written as 2 $argIs$ assertions. Instead of writing full rule, the constraint for the argument of *subclass* can be written simply as $argIs(subclass, 1, Class)$. To make this hold for all the combinations of predicates (not just *subclass* from example), we can re-phrase the rule to be general, and also define the $argIs$ predicate.

Definition 4.6 (predicate "$argIs$"). Predicate argIs(x,y,z) denotes that the $y-th$ argument of predicate x, must be an instance of z. For example, asserting $argIs(subclass, 1, Class)$, states that the first argument of predicate *subclass* must be an instance of $Class$. This is enforced by the following constraint rule:

$$\begin{gathered} \forall P \forall n \forall x_1 ... x_n \forall C_{1...m} : \\ ((arity(P, n) \wedge P(x_1, ..., x_n)) \implies (is(x_1, C_{1...m}) \wedge ... \wedge is(x_n, C_{1...m})) \\ \iff \\ argIs(P, 1, C_{1...m}) \wedge ... \wedge argIs(P, n, C_{1...m})) \end{gathered} \tag{4.13}$$

These definitions allow us to use simple $argIs$ assertions, instead of complicated rules. For the cases, when the arguments shouldn't be instances of a $Class$, but its subclasses, we can define similar predicate and its constraint rules also fro $argClass$:

Definition 4.7 (predicate "$argClass$"). Predicate $argClass(x, y, z)$ denotes that the $y-th$ argument of predicate x, must be a subclass of z. For example, asserting $argClass(servesCuisine, 2, Restaurant)$, states that the first argument of predicate $servesCuisine$ must be a subclass of $Restaurant$. This is enforced by the following constraint rule (similar

as for $argIsa$, but for $argClass$):

$$\begin{gathered}\forall P \forall n \forall x_1 ... x_n \forall C_{1...m} : \\ ((arity(P,n) \wedge P(x_1,...,x_n)) \implies (subclass(x_1, C_{1...m}) \wedge ... \wedge subclass(x_n, C_{1...m})) \\ \iff \\ argClass(P,1,C_{1...m}) \wedge ... \wedge argClass(P,n,C_{1...m}))\end{gathered} \tag{4.14}$$

Before we can assign argument constraints to our existing predicates and have all the KB valid, we need to define a special class $Number$:

Definition 4.8 (class $Number$ and its instances). All natural numbers are instances of the class $Number$. Formally, this can be asserted into a KB as:

$$\forall x \in \mathbb{N} : is(x, Number) \tag{4.15}$$

This now allows us to use $argIs$ and $argClass$ predicates instead of complicated rules, to define types of arguments inside any predicate used in the KB. By using these two newly defined predicates, we can now proceed to assert the argument limits of the $argIsa$ predicate itself:

$$argIs(argIs, 1, Predicate) \wedge argIs(argIs, 2, Number) \wedge argIs(argIs, 3, Class) \tag{4.16}$$

We can see that the first argument must be an instance of $Predicate$, which holds in all three cases ($argIs$ is a first argument of the Assertion 4.16 above), since it was defined in Assertion 4.5. Similarly, the second argument is a valid number, in all three cases as defined in Definition 4.8. Also the third arguments ($Predicate$, $Number$,$Class$), are all instances of the $Class$ as defined in Assertion 4.6, so the assertion 4.16 can be asserted and it doesn't invalidate itself through argument constraint rule (Constraint 4.13).

Similary, we can now proceed to define the rest of our predicates. Starting with the most similar $argClass$:

$$\begin{gathered}argIs(argClass, 1, Predicate) \wedge argIs(argClass, 2, Number) \\ \wedge argIs(argClass, 3, Class)\end{gathered} \tag{4.17}$$

Since we didn't yet assert any direct $argClass$ constraint, this predicate at this point defines the constraints, without any danger to invalidate our current KB.

Continuing with $subclass$. On the Definition 4.5, we can see that this predicate defines sub-class relationships between the classes. For this reason it makes sense to only allow instances of $Class$ for its arguments:

$$argIs(subclass, 1, Class) \wedge argIs(subclass, 2, Class) \tag{4.18}$$

Same as with the Assertion 4.17, we didn't yet assert any direct assertions about something being a sub-class of something, this assertion doesn't affect yet the validity of our KB, while prevents future assertions of $subclass$ predicate on anything but $Class$ instances.

For the $arity$ predicate, we can check our existing assertions (4.8, 4.9), and see that as the first argument we always have an instance of $Predicate$, while as the second argument we have an instance of a $Number$. According to this, it serves our purpose and is safe to limit the arguments of $arity$ to:

$$argIs(arity, 1, Predicate) \wedge argIs(arity, 2, Number) \tag{4.19}$$

We can now proceed to define our last (*is*) predicate constraints, which is a bit more complicated. If we look at our existing *is* assertions (4.4, 4.5, 4.15), we can see that as a second argument we always have an instance of a *Class* (*Predicate*, *Class*), but for the first argument we can actually put in anything (*is*, *Class*, *Predicate*, *Number*, $\mathbb{N}$). From instance of a *Class*, instance of a *Predicate*, to an instance of a *Number*. For the second argument we can immediately assert

$$argIs(is, 2, Class) \tag{4.20}$$

In order to be able to say that some argument can be anything, we followed a Cyc example and introduce term *Something*, first mentioned in the set of our terms in Assertion 4.1. We set this as the constraint for the first argument of *is* predicate:

$$argIs(is, 1, Something) \tag{4.21}$$

But this assertion above (4.21), invalidates the correctness of all of our *is* assertions, since none of the current first arguments are instances o *Something* (see Assertions 4.4, 4.5 and 4.15). To fix this, we need to be able at least to say that things are instances of *Something* ($is(x, Something)$). According to the *is* argument constraint assertion above (4.20), *Something* must be an instance of a *Class*. So we define it as so:

$$is(Something, Class) \tag{4.22}$$

Now, as a consequence of this, we could assert for all of the arguments that are used in *is* (*is*, *Class*, *Predicate*, *Number*, $\mathbb{N}$), that they are an instances of the *Something*. This would be highly unpractical, since we would need to do this for every future constant to be used by *is* predicate (especially unpractical for infinite number of $\mathbb{N}$).

Instead, since we know that *Something* is an instance of a *Class*, we can use it in our *subclass* assertions (a consequence of constraint Assertion 4.18) and state that *Class* is a subclass of *Something*:

$$subclass(Class, Something) \tag{4.23}$$

A consequence of this assertion is (because of inference rule 4.10), that every sub-class of anything that exists in our KB (because we can only use instances of *Class* in *subclass* assertions), is a sub-class of *Something* as well. This doesn't yet seem to help us make our 1st *is* predicate arguments instances of *Something*, but it will, after we address another weakness in our current KB. Consider the continuation of the example we started in *subclass* definition (Definition 4.5, where a terrier is a sub-class of dog, and dog is a sub-class of animals ($subclass(Dog, Animal)$, $subclass(Terrier, Dog)$). If we introduce an instance of the class *Terrier*, let's say, a real dog named "Spot" ($is(Spot, Terrier)$), we can see that there is a logical problem, since Spot is a terrier in our KB, but not a dog, or even an animal (there is nothing to support $is(Spot, Animal)$ assertion. This can be fixed by introducing the following Inference Rule:

$$\forall x \forall y \forall z : ((is(x, y) \wedge subclass(y, z)) \implies is(x, z)) \tag{4.24}$$

This rule is basically saying, that if there is an instance of a class, and this class is a sub-class of another class, then this instance is also an instance of the other class. Now, this inference rule (4.24), together with the fact that *Class* is a sub-class of *Something*, and the *subclass* transitivity inference rule (4.10), makes everything that is instance of a class, or its sub-class (which is everything in our KB), also an instance of *Something*, and thus proving the assertion 4.21 correct and consequently our KB fully consistent again.

At this point our Upper Ontology is defined (it is visually presented on Figure 4.3) and ready to build upon as will be described in the next chapters. Since Curious Cat main implementation is based on Cyc, and it was inspired by the way Cyc ontology is constructed and being used, this upper ontology reflects the main part of Cyc upper ontology (see Chapter 5, implementation on how this formalization maps to Cyc). While our upper ontology logical definitions and formalizations are strongly influenced by Cyc, the approach is general and not bound to any particular implementation, and our notation reflects but is not tightly bound to that of Cyc. For example, the usage of *argIs* and *argClass* can be replaced by *domain* and *range* when using a RDF schema, such as was done in our RDF prototype implementation(Bradeško, Moraru, et al. 2012), or a completely custom constraints can be used in specific ontologies, so this upper ontology is more of a guidance and a tool to be able to explain approach, than a fixed ontology that needs to be implemented.

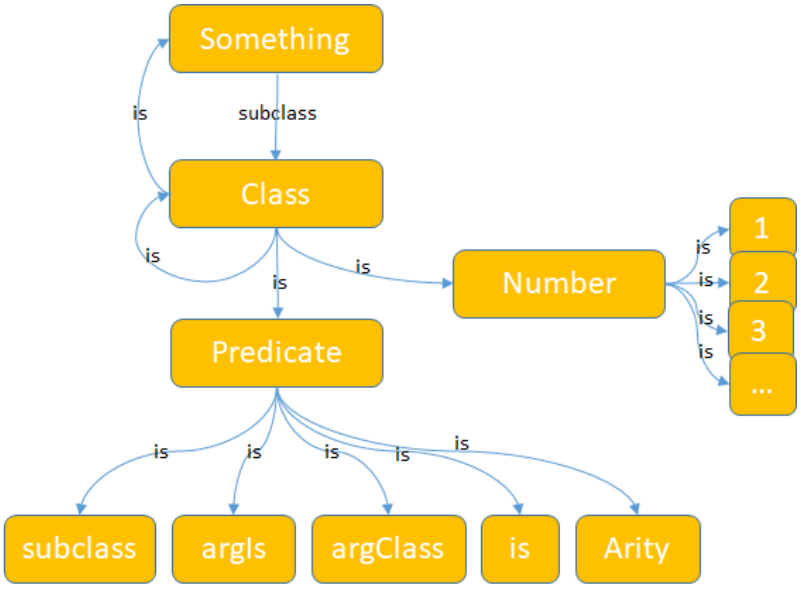

Figure 4.3: Upper ontology terms, with 'is' and 'subclass' relations.

4.2.2 Existing Knowledge

This sub-chapter extends our upper ontology example with additional knowledge that allows us to explain the system through the examples given in Table 4.1 and Table 4.2.

As also visible in the Architecture schematic (Figure 4.1, KB with background knowledge is one of the most crucial elements of proposed approach. It serves both, as the driving force behind the source of the questions, since with the help of contextual knowledge triggers the inference to produce logical queries for the missing parts of the knowledge. At the same time it is the drive behind the proactive user interaction, starting either as a question or a suggestion. Finally, the background knowledge is also used for validation of answered questions. If the answers are not consistent according to the existing KB, users are required to re-formulate, or repeat the answer.

The main *Curious Cat* implementation, uses an extended full Cyc ontology and KB, similar to that released as ResearchCyc, as a common sense and background knowledge base. This is far too big (millions of assertions), to be explained in any detail here with the general approach (it is explained in more detail in Chapter 5, Implementation). In this chapter we define only the concepts and predicates and thus construct the minimal

example KB that is necessary for explaining the proposed system.

First we introduce the set of new concepts that we need for the food part of the ontology. These concepts are defined on top of the existing Upper Ontology, so the new parts of the KB should maintain the consistency of the already described parts. The set of terms S_{Food}, needed to describe this part of the KB is as follows:

$$S_{Food} = \{FoodOrDrink, Food, Bread, Baguette, Drink, Coffee\} \tag{4.25}$$

Where each of these terms is an instance of a $Class$:

$$\forall x \in S_{Food} : isa(x, Class) \tag{4.26}$$

Then, these classes are connected into a class-hierarchy, which is done with a *subclass* predicate:

$$\begin{gathered} subclass(Drink, FoodOrDrink) \wedge subclass(Coffee, Drink) \wedge \\ subclass(Food, FoodOrDrink) \wedge subclass(Bread, Food) \wedge \\ subclass(Baguete, Bread) \end{gathered} \tag{4.27}$$

Which is also presented graphically on Figure 4.4 below.

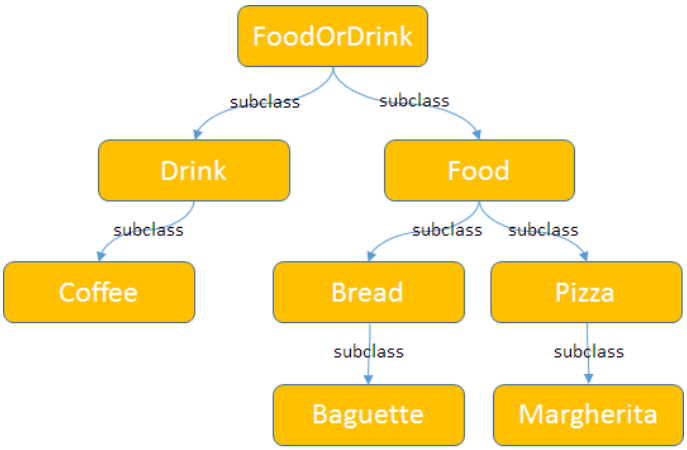

Figure 4.4: Hierarchy of food related terms, specified by subclass predicates.

The second part of ontology (KB) constructing knowledge about places consist of the following terms:

$$S_{Place} = \{Place, PublicPlace, PrivatePlace, Restaurant\} \tag{4.28}$$

Where each of these terms is also an instance of a $Class$:

$$\forall x \in S_{Place} : isa(x, Class) \tag{4.29}$$

Then, these classes are connected into a class-hierarchy, which is done with a *subclass* predicate:

$$\begin{gathered} subclass(PrivatePlace, Place) \wedge subclass(PublicPlace, Place) \wedge \\ subclass(Restaurant, PublicPlace) \wedge subclass(Home, PrivatePlace) \end{gathered} \tag{4.30}$$

And, in order to have some "real-world" knowledge, we also add an example instance of a real restaurant represented as term $Restaurant1$.

$$is(Restaurant1, Restaurant) \tag{4.31}$$

This is all presented graphically on Figure 4.5 below.

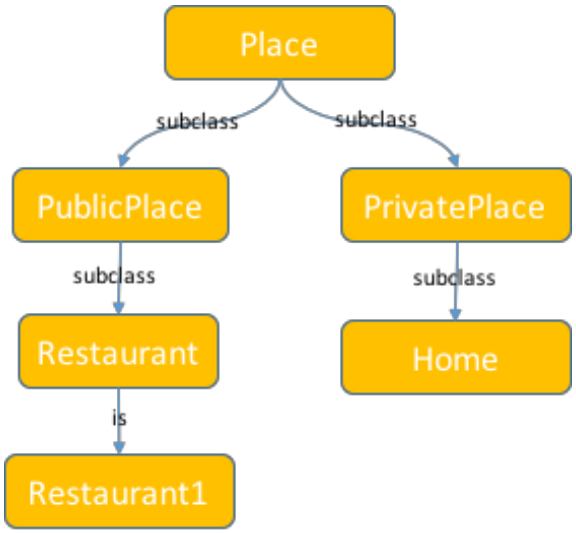

Figure 4.5: Hierarchy of place related terms, specified by subclass predicates.

Then the last part (excluding predicates which are defined separately), of our *Existing Knowledge* consist of the following terms:

$$\begin{aligned} S_{Rest} = \{&Service, WirelessService, Vehicle, Car, Animal, Duck,\\ &Human, User, User1\} \end{aligned} \tag{4.32}$$

Where each of these terms is also an instance of a $Class$:

$$\forall x \in S_{Rest} : isa(x, Class) \tag{4.33}$$

And a class-hierarchy:

$$\begin{aligned} &subclass(Car, Vehicle) \wedge subclass(WirelessService, Service) \wedge\\ &subclass(Duck, Animal) \wedge subclass(Human, Animal) \wedge\\ &subclass(User, Human) \end{aligned} \tag{4.34}$$

And an instance of the $User$ class representing one CC user.

$$is(User1, User) \tag{4.35}$$

Which is also presented graphically on Figure 4.6 below.

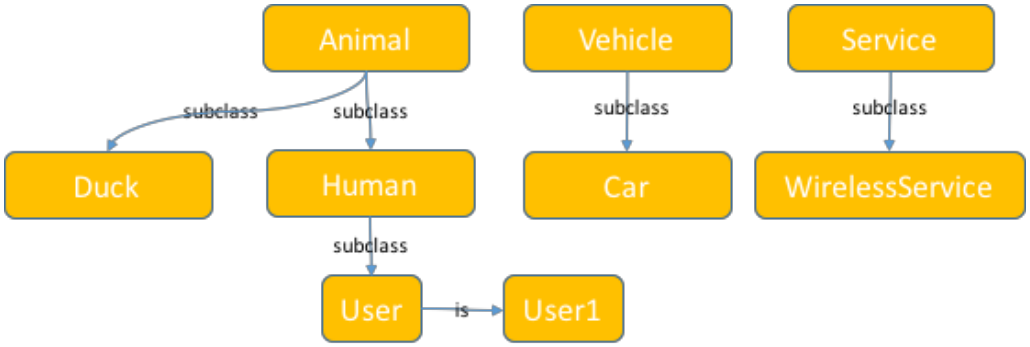

Figure 4.6: Hierarchy of the rest of the terms from our *existing knowledge*.

We can see that all these new terms add on top of the existing set of terms from upper ontology ($S_{Constants} = S_{upper} \cup S_{Food} \cup S_{Place} \cup S_{Rest}$).

For defining the predicates, we need a bit more detailed definitions, since we also want to represent the constraints which will serve for the inference engine to check the validity of the answers (as explained in the definitions 4.6 and 4.7).

Definition 4.9 (predicate *menuItem*). Predicate $menuItem(x, y)$ denotes that place x has a menu item y on its menu.Formally it is defined as:

$$\begin{gathered} is(menuItem, Predicate) \wedge \\ arity(menuItem, 2) \wedge \\ argIs(menuItem, 1, Restaurant) \wedge \\ argClass(menuItem, 2, FoodOrDrink) \end{gathered} \tag{4.36}$$

As we see defined above (assertion 4.36), this predicate constraints allow us to only use instances of class *Restaurant* for the first argument, and sub-classes of class $FoodOrDrink$ for the second. Example of this is the following assertion that we use to have an example restaurant instance in our KB (to continue the definition of Assertion 4.31), saying that Restaurant1 has coffee on the menu:

$$menuItem(Restaurant1, Coffee) \tag{4.37}$$

Besides the items on the menu, to explain our examples, we also need a way to tell that a place provides some services and that user ordered something while visiting a restaurant.

Definition 4.10 (predicate *providesServiceType*). Predicate $providesServiceType(x, y)$ denotes that place x provides service y. Formally it is defined as:

$$\begin{gathered} is(providesServiceType, Predicate) \wedge \\ argIs(providesServiceType, 1, Place) \wedge \\ argClass(providesServiceType, 2, Service) \end{gathered} \tag{4.38}$$

Definition 4.11 (predicate *userOrdered*). Predicate $userOrdered(x, y, z)$ denotes that user x as part of his visit y ordered something from the menu z. Formally it is defined as:

$$\begin{gathered} is(userOrdered, Predicate) \wedge \\ arity(userOrdered, 3) \wedge \\ argIs(userOrdered, 1, User) \wedge \\ argIs(userOrdered, 2, Visit) \wedge \\ argClass(userOrdered, 3, FoodOrDrink) \end{gathered} \tag{4.39}$$

As defined in the Assertion 4.38, *providesServiceType* constraints allow us to only use instances of class *Place* (Public,Private, Home,Restaurant for the current stage of the KB) for the first argument, and sub-classes of class *Service* for the second. Similarly, the predicate *userOrdered* can only take instances of *User* for the first argument, instances of *Visit* for the second, and subclasses of *FoodOrDrink* for the third.

At this point we can also add an inference rule, stating that if user ordered something at some place, then this same thing is on the menu at this place as well:

$$placeVisit(p, v) \wedge userOrdered(u, v, x) \wedge \implies menuItem(p, x) \tag{4.40}$$

Definition 4.12 (predicate *userLocation*). Predicate $userLocation(x, y)$ denotes that user x is located at place y. Formally it is defined as:

$$\begin{aligned} &is(userLocation, Predicate) \wedge \\ &argIs(userLocation, 1, User) \wedge \\ &argIs(userLocation, 2, Place) \end{aligned} \tag{4.41}$$

As defined in the Assertion 4.41, *userLocation* constraints allow us to only use instances of class *User* for the first argument, and instances of *Place* as a second.

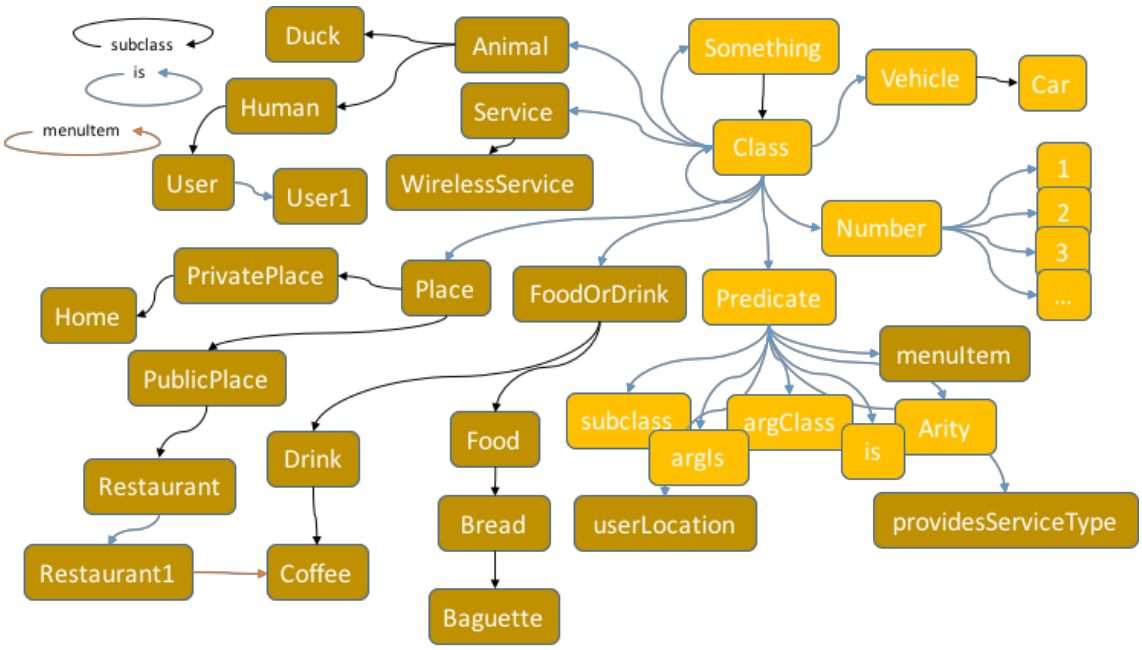

Figure 4.7: Current "existing knowledge" on top of upper ontology

At this point we have a formal KB structure representing a minimal *existing knowledge* required to explan the examples from Table 4.1 and Table 4.2. The KB is graphically presented on the Figure 4.7, where the upper ontology is presented in lighter color and new KB parts in darker. Due to lack of space, the only relations are of *is* (represented by blue arrows), *subclass* (represented with black arrows) and *menuItem* (orange arrow) predicates.

4.2.3 KA Knowledge

In the previous two sections we defined the upper ontology (Section 4.2.1) and then using its vocabulary to define the preexisting knowledge (Section 4.2.2). This will suffice to support the explanation of the proposed KA approach. Similar as with the other parts of the KB, listing full set of *Curious Cat* KA rules would not fit in the paper. Instead we define an example set of the KB, sufficient for describing the approach and keeping the explanation as simple as possible.

As a starting point, we need to define a main KA meta-class $Formula$, which's is a special class (like $Number$).

Definition 4.13 (class $Formula$ and its instances). All logical formulas (assertions, queries) are instances of the class $Formula$. Formally, this can be represented as:

$$\forall P \forall x_{1...n} : is(P(x_1, ..., x_n), Formula) \tag{4.42}$$

Basically all of the content of the KB and queries are instances of $Formula$. For example, assertion $userLocation(User1, Ljubljana)$ is an instance of $Formula$ and consequentially the statement $is(userLocation(User1, Ljubljana), Formula)$ is true.

Then we need to define additional KA meta predicates:

Definition 4.14 (predicate "$known$"). Special meta-predicate $known(x)$ denotes that the formula x can be proven in the KB. This predicate is non-assertible, meaning that it cannot be used to add things into the KB, but it can be used in inference rules. For example, since the Assertion 4.22 is already asserted in the KB and true, the query/unassertible statement $known(is(Something, Class))$ is also true. The predicate is formally defined as follows:

$$is(known, Predicate) \wedge arity(known, 1) \wedge argIs(known, 1, Formula) \tag{4.43}$$

In a similar fashion as predicate $known$, we define $unknown$ predicate.

Definition 4.15 (predicate "$unkknown$"). Special meta-predicate $unknown(x)$ denotes that the formula x can **not** be proven in the KB. This predicate is non-assertible, meaning that it cannot be used to add things into the KB, but it can be used in inference rules. Building on the example given in the Definition 4.14 ($known$ predicate), the query $unkown(is(Something, Class))$ is **not true**, since the assertion exist an is known. On the other hand, $unknown(is(Human, Duck))$ is true, since there is no knowledge in the KB which would support the encapsulated is statement. The predicate is formally defined as follows:

$$is(unknown, Predicate) \wedge arity(unknown, 1) \wedge argIs(unknown, 1, Formula) \tag{4.44}$$

Similarly, we need a way to tell whether two concepts of formulas are the same.

Definition 4.16 (predicate "$equals$"). Predicate $equals(x, y)$ denotes that the concept or formula x is exactly the same as y. For example, $equals(Dog, Dog)$ is true, while $equals(Dog, Cat)$ is not. The predicate is formally defined as follows:

$$\begin{gathered} is(equals, Predicate) \wedge arity(equals, 2) \\ \wedge argIs(equals, 1, Something) \wedge argIs(equals, 1, Something) \end{gathered} \tag{4.45}$$

And now one of the main KA predicates, which is used to provide the CC system with the formulas which need to be converted to NL and presented to the user.

Definition 4.17 (predicate "*ccWantsToAsk*"). One of the main KA predicates written as $ccWantsToAsk(x, y)$, denotes that the *Curious Cat* system wants to ask user x a question represented by the formula y. For example, assertion

$$ccWantsToAsk(User1, userLocation(User1, x))$$

tells CC system to ask user the question $userLocation(User1, x)$, which, after it goes through logic to NL conversion (subsection 4.4.1) is paraphrased as "Where are you?", as hinted in the step 1 in the Table 4.1. The predicate is formally defined as follows:

$$\begin{gathered} is(ccWantsToAsk, Predicate) \wedge arity(ccWantsToAsk, 2) \wedge \\ argIs(ccWantsToAsk, 1, User) \wedge argIs(ccWantsToAsk, 2, Formula) \end{gathered} \tag{4.46}$$

Now, after we have supporting predicates, we can define an example of KA rule, KA rules can written specifically to enable the production of questions for a narrow context, or generally, covering broad scope of questions as the following:

$$\begin{gathered} \forall i_1 \forall c \forall i_2 \exists s_2 \forall u : is(i_1, c) \wedge P(i_1, s) \wedge is(i_2, c) \wedge unknown(P(i_2, s_2)) \\ \Longrightarrow \\ ccWantsToAsk(u, P(i_2, \$x)) \end{gathered} \tag{4.47}$$

This rather complicated material implication causes generation of question intents whenever there is an instance of a class that was used in an arity 2 predicate, and there is another instance of the same class which does not have any assertion using this predicate. When the antecedent of this rule is true, then the consequent is a *ccWantsToAsk* predicate representing an open ended logical query (NL question) asking for the things that can fulfill the predicate P for instances of aforementioned class.

If we take our example KB (Figure 4.7) and imagine we assert an additional instance of a *Restaurant* (Restaurant2: "Joe's Pizza", as was done in the example conversation in Table 4.1, step 2, the premises of the rule can get satisfied like this:

$$\begin{gathered} is(Restaurant1, Restaurant) \wedge menuItem(Restaurant1, Coffee) \wedge \\ is(JoesPizza, Restaurant) \wedge unknown(\exists s_2 : menuItem(JoesPizza, s_2)) \end{gathered} \tag{4.48}$$

Because all of the above premises are true, including the *unknown* part (there is no support in the KB for the formula inside *unknown* predicate - see Definition 4.15), this rule in our KB up to now produces the following consequent:

$$ccWantsToAsk(User1, menuItem(JoesPizza, x)) \tag{4.49}$$

representing an intent of CC system to ask user the question $menuItem(JoesPizza, x)$, which converted to NL comes out as "What is on the menu in Joe's Pizza", which matches the step 5 from the Table 4.1. In the general KA rule example (Assertion 4.47), we can notice that one of the variables ($\$x$) is not presented with standard variable naming (x, y, z), but is marked with a prefix \$, which is a marker for the inference engine to not treat this as a standard variable and try to bind it to values inside KB. Instead variables marked with \$, are treated as normal concepts until they are asserted into the KB or used as in the KB as a query.

The rule 4.47 effectively detects when there is an instance in the KB that does not have some kind of information that other instances have, and then it causes the system to intend to ask about it, if and when it has a suitable opportunity (e.g. a suitable interaction context). The rule described, is an example of the general rule that can produce

the apparent curiosity of the system using nothing but the existing background or newly acquired knowledge, whatever that may be.

In very large knowledge bases, general rules like this can produce many questions, including, in some cases, many irrelevant ones. To mitigate this, the proposed system must have additional rules that can suppress questions on some predicates, or for whole parts of the KB. Simples example for this is to introduce $omitPredicate(x)$ predicate, with a constraint $argIs(omitPredicate, Predicate)$, which can be then either used int rules like 4.47 as one of the conjunction premises using *unknown* predicate, or the resulting $ccWantsToAsk$ sentences can be filtered with additional rules, or by *procedural component*, removing questions using *omitted* predicates.

While we defined rule 4.47 in some detail here to show the possibilities of the approach, we will explain the rest of the system through simpler examples for easier understanding. For example, the narrower rule which produces the same step 5 of example shown in Table 4.1 can also be defined as follows:

$$\forall x \forall u : is(x, Restaurant) \implies ccWantsToAsk(u, menuItem(x, \$y)) \tag{4.50}$$

In this case, the system would always produce the *menuItem* questions for all, existing restaurants, regardless of whether this knowledge exists already or not, it would always aks more. But only for *menuItems*, as the rule is not general and would never come out of new type of questions as does rule 4.47.

We can see now that the KA or "curiosity rules" can span from very general, to very specific. General rules can automatically trigger on almost any newly asserted knowledge, while specific ones can be added to fine-control the responses and knowledge we want to acquire. How specific or general the rule will be, is simply controlled by the number and content of the rule premises.

4.3 Context

As mentioned in section 2.2, in order to be able to ask relevant questions which users can actually answer, and at the same time maintain their interest, the context of the user is of crucial importance. For this reason, a considerable part of our KB content is user context, which can be used with the KA rules as the rest of the knowledge. One example of contextual knowledge is the $userLocation(x, y)$ predicate (Definition 4.12), which holds the information combined from mobile sensors mining (see the definition of $probableUserLocation$ below) and the KA process. Besides $userLocation$, there is a lot of additional information on the user that the proposed system uses to coming up with personalized questions.

4.3.1 Mined Context

For the system to be able to address users at the right time and about the right knowledge, it makes sense to understand user and his/her current context as much as possible. While there is infinite amount of things that can contribute to contextual knowledge, the central and most important piece in our approach is user's location and the duration of stay at this location.

While for this (CC) approach it doesn't matter how exactly the current location and other contextual raw data is acquired, since the year 2007 (release of iPhone and Nokia N95) it is the easiest to acquire it from the user's mobile phones[2]. As we hinted already in the

[2]With the prior user's consent.

subsection 3.5.1, we use a modified *stay-point detection* algorithm, similar to the approach taken by Kang et al. (2005) (algorithm 3.1). Here we will describe the mining algorithm only briefly, to be able to explain the approach. It is described in more detail in section 5.4.2. These algorithms are all implemented in our *procedural component* (seesection 4.1, depicted on Figure 4.1).

This algorithm takes raw GPS readings as input, and clusters them into visits (staypoints) and paths (moves between the staypoints), based on two thresholds: the time (T_t) the user needs to stay inside the perimeter (T_P), which is the second threshold. The staypoint is defined with the coordinates (of the users mobile device) that all lay inside a given perimeter for at least minimum time: $r(lat, lon) < T_p \wedge t(lat, lon) > T_t$. The result of this algorithm is one coordinate (lat, lon), time of arrival and the duration of stay, which can be directly used by our approach (see 4.18 below). This simple algorithm proves to be robust to the usual GPS signal, which is not always returning the same coordinates for the same location and is also lost during the time the user is indoors, or nearby tall buildings or trees.

After we have the coordinates, time of arrival and current duration of stay, we can use this info to get additional information about the place, most easily from one of the online APIs (Foursquare, Factual places, Google Places). This usually return multiple results, where the less likely ones need to be filtered out. This is described in detail in the implementation (section 5.4.2), related work (Section 3.5.3) and also in the related papers (Mamei 2010; Bradesko et al. 2015).

Let's say the user moves from the previous location. This is detected by the algorithm (move), which causes the *Procedural Component* to remove previous $userLocation(x, y)$ assertion for this user, since his/her location is not known anymore. Then the SPD algorithm (section 5.4.2 is monitoring the user, and once it detects that he/she stopped at some location for a bit, it retrieves the most likely place from public APIs. After this, the enriched and relevant information can be added back to the KB. For this we need to define (add) a few more pieces of the KB.

Definition 4.18 (predicate "*probableUserLocation*"). Predicate $probableUserLocation(x, y)$ denotes that the user x is most likely at the location y. It is "probable" because it is automatically inferred by the ML algorithm and we still want the user to confirm it. Formally the predicate is defined as:

$$\begin{aligned} &is(probableUserLocation, Predicate) \wedge \\ &argIs(probableUserLocation, 1, User) \wedge \\ &argIs(probableUserLocation, 2, Place) \end{aligned} \tag{4.51}$$

Definition 4.19 (class "*Visit*" and its instances). Class $Visit$ represent a concept of a visit. Each instance represent one particular visit, which can be used in other assertion to add more data to it or to link it to other concepts. For example see Assertion 4.59.

Definition 4.20 (predicate "*userVisit*"). Predicate $userVisit(w, x, y, z)$ denotes that the user w did a visit x at the time y, where visit duration was z. This predicate is used to be able to link users to their visits with some additional data like time of arrival (sa unix time), and the duration of stay (in seconds), which proved

to be an important part of the context. Formally the predicate is defined as:

$$\begin{gathered} is(userVisit, Predicate) \wedge \\ arity(userVisit, 4) \wedge \\ argIs(userVisit, 1, User) \wedge \\ argIs(userVisit, 2, Visit) \wedge \\ argIs(userVisit, 3, Number) \wedge \\ argIs(userVisit, 4, Number) \wedge \end{gathered} \tag{4.52}$$

Definition 4.21 (predicate "*placeVisit*"). Predicate $placeVisit(x, y)$ denotes that the place x was visited as part of the visit y. This predicate is used to link visits to particular places. Formally the predicate is defined as:

$$\begin{gathered} is(placeVisit, Predicate) \wedge \\ arity(placeVisit, 2) \wedge \\ argIs(placeVisit, 1, Place) \wedge \\ argIs(placeVisit, 2, Visit) \wedge \end{gathered} \tag{4.53}$$

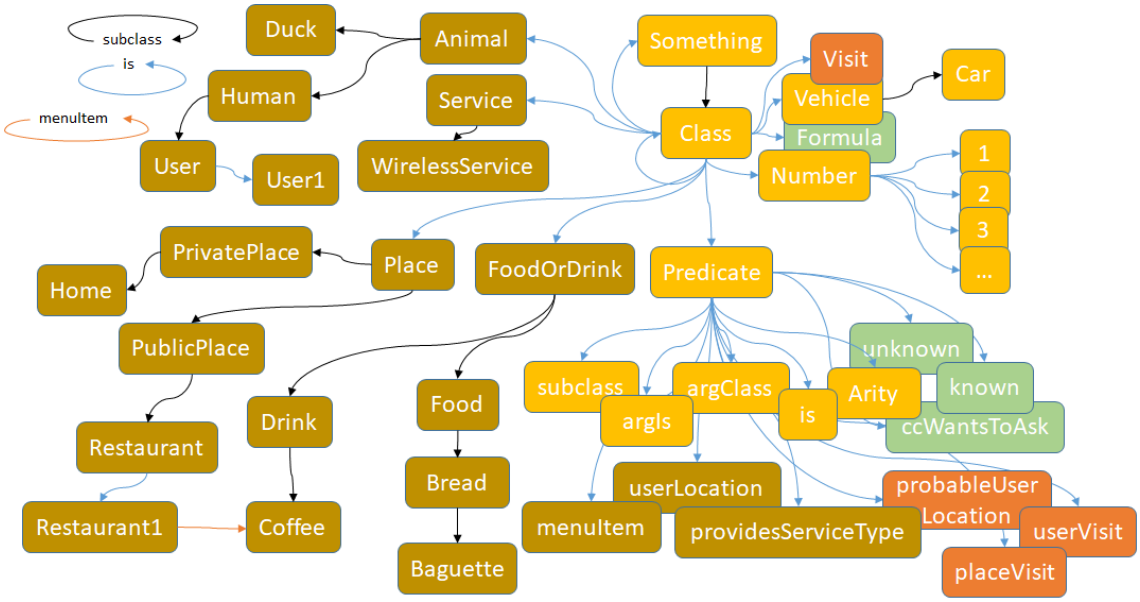

Figure 4.8: Current KB with newly added "context support knowledge" marked with orange.

Now, after we defined the relevant predicates (see Figure 4.8), we can continue building the example. Let's say that the most likely place returned by SPD (section 5.4.2), is "Joe's Pizza" (our usual example from Table 4.1). The *Procedural Component* creates a new concepts $JoesPizza$ and $Visit1$ (it has number 1 since it is the first instance in the KB), and adds them into the KB:

$$is(JoesPizza, Place) \tag{4.54}$$

and

$$is(Visit1, Visit) \tag{4.55}$$

After this, the system adds more contextual data to these concepts. Let's say the user arrived to the place on 19. Sept 2017 at 3pm, and is at the moment there for 3 minutes. This converts into the following assertions being added, while removing possible prior assertions for the same user and visit combination:

$$probableUserLocation(User1, JoesPizza) \tag{4.56}$$

and

$$userVisit(User1, Visit1, 1505746800, 180) \tag{4.57}$$

Then, continuing our example, whenever te system gets a new GPS coordinate (let's say in 10 seconds), and consequently our algorithms calculate context update, the system deletes the Assertion 4.57, and updates it with new data:

$$userVisit(User1, Visit1, 1505746857, 190) \tag{4.58}$$

These updates (even if just duration of visit seconds) are causing the CC system inference engine to re-evaluate the rules connected to relevant predicates and thus possibly produce new consequents in the shape of new questions or comments, which appears to the users as pro-activity, since they get new relevant questions even if not interacting with the system. This simple contextual knowledge(presented with Assertions 4.56 and 4.58) is enough to be able to show how to make the system produce the examples provided in Table 4.1 and Table 4.2.

At this moment careful reader might notice that the predicate *placeVisit* was not used in the assertions. The predicate is not yet asserted, because at this time it is not yet known for sure, whether the user is exactly at this place. The *probableUserLocation* is only used to trigger questions for the predicate *userLocation* (see definition 4.12). Only after this is answered, we can be 100% sure that this is the exact place, and, besides the *userLocation* (see assertions, the system can also assert

$$placeVisit(JoesPizza, Visit1) \tag{4.59}$$

and thus permanently link the user's 19. Sept 2017 visit to the place Joe's Pizza.

As described and defined above (assertion 4.58), we can use *userVisti* predicate to specify a length of stay of user at particular place. This allows the system to react on this information, and produce time dependent questions and decisions. But in order to be able to do it more generally (as oposed to react only for the exact second when user satayed somewhere), we define a new predicate *greaterThan*.

Definition 4.22 (predicate "*greaterThan*"). Predicate $greaterThan(x, y)$ denotes that x has a greater value than y. Meaning, this predicate is only true when $x > y$. For example, stating $greaterThan(10, 5)$ is true.

This predicate can be then used in rules that react based on some numeric values like times, number of votes, number of answers, etc.

4.3.2 Acquired context

Besides the context mined automatically, the system can also use internal context, which is a specific set of KA rules and KB knowledge acquired from the user himself, or other users, and is directly relevant to the the him/her. By considering the knowledge acquired from the user, we improve relevance of asked questions. This knowledge is obtained by

asking the user specific questions about himself, such as, the languages spoken, profession, interests, preferred food, etc., represented by the example predicates below:

$$userAge(x, y)$$
$$userSpeaksLanguage(x, y)$$
$$userInterest(x, y)$$

or, a lot of knowledge can also be inferred already even by questions asked in our examples (Table 4.1), resulting in many instances of $Visit$ ($Visit1, Visit2, Visit3, ...$) and *orderedFood* predicates. Specific rules or analytics provided by the *Procedural Component* can infer things like the following examples:

$$userHome(x, y)$$
$$userHomeCity(x, y)$$
$$userHomeCountry(x, y)$$
$$userLikesFood(x, y)$$
$$...$$

The knowledge gathered in this way, is additional to the mined context and can be used by the rules to better identify the users who will actually be able to answer particular questions.

4.4 NL to logic and logic to NL conversion

In order to interact with the user, it is not sufficient to form the knowledge acquisition questions in logical form using an inference engine and KA rules, as we had seen up to now. These formulas are understandable by knowledge engineering and math experts, but are not at all appropriate for a direct use in general KA using crowdsourcing from the general population. For this reason, the logical formulas of the sentences and questions need to be translated to natural language to be presented to the user. Similarly, as the user is providing some of the answers in natural language these at least to some extent have to be transformed from natural language, into their logical form. This means that in addition to the knowledge itself the KB should include natural language description of the knowledge units, or the natural language generation capabilities must be provided by an external service (the letter is the case in our example with Umko KB (Bradesko et al. 2015) and early crowdsourcing approach (Bradeško, Moraru, et al. 2012).

Because natural language generation and conversion is not the main focus of this paper, we present here only a simplified version which explains the basic concepts involved. The actual Curious Cat implementation is based on Cyc NL(Baxter et al. 2005) for generation, and SCG(Schneider et al. 2015) for the NL to logic. It consists of more than 90 predicates and rules beyond those in the baseline Cyc system to handle language generation. This is described in more detail in the Implementation chapter (Chapter 5, section 5.4.2).

4.4.1 Logic to NL

When the system already employs an extensible KB, each of the concepts in the KB can be named using a standard textual string, and predicates can have attached a knowledge on how to represent themselves in natural language. For this, we first need to add a concept of *String* on top of our *upper ontology*.

Definition 4.23 (class *String* and its instances). All written texts (strings), marked with double quotes, are instances of the class *String*. Formally, this can be asserted into a KB as:

$$\begin{gathered}\mathbb{S} = \{x | \forall y : x = "y"\} \\ \forall x \in \mathbb{S} : is(x, String)\end{gathered} \tag{4.60}$$

For example, the following is a correct logical statement and is also true:

$$is(\text{"This is string"}, String)$$

Now we can also define the *name* and *namePlural* predicates, which are used to add to concepts their name string representations.

Definition 4.24 (predicate "*name*"). Predicate $name(x, y)$ denotes that the concept x has a name, or string representaion given as y. For example, asserting $name(JoesPizza, \text{"Joe's Pizza"})$, states that the string representation for the concept $JoesPizza$ is "Joe's Pizza". Formally the predicate is defined as:

$$\begin{gathered}is(name, Predicate) \wedge \\ arity(name, 2) \wedge \\ argIs(name, 1, Something) \wedge \\ argIs(name, 2, String)\end{gathered} \tag{4.61}$$

Definition 4.25 (predicate "*namePlural*"). Predicate $namePlural(x, y)$ denotes that the concept x has a plural version of its name representaion given as a string y. For example, asserting $namePlural(Pizza, \text{"pizzas"})$, states that the plural version of string representation for the concept $Pizza$ is "pizzas". Formally the predicate is defined as:

$$\begin{gathered}is(namePlural, Predicate) \wedge \\ arity(namePlural, 2) \wedge \\ argIs(namePlural, 1, Something) \wedge \\ argIs(namePlural, 2, String)\end{gathered} \tag{4.62}$$

And similarly, *nlPattern* predicate, attaching a NL generation information to other predicates:

Definition 4.26 (predicate "*nlPattern*"). Predicate $nlPattern(x, y)$ denotes that the predicate x has a NL pattern that can be used in sentences given as y. For example, asserting $nlPattern(is, \text{"\$1 is a \$2"})$, states that the predicate is, can be represented in natural language as the name of first argument, followed by the string "is a", and then followed by the name of the second argument. Based on our example NL engine, the arguments are marked with the "$" sign, followed by the number. When some of the arguments are missing, this represents an interrogative mode. For example $nlPattern(is, \text{"What is \$1"})$. See below for more details. Formally the predicate is defined as:

$$\begin{gathered}is(nlPattern, Predicate) \wedge \\ arity(nlPattern, 2) \wedge \\ argIs(nlPattern, 1, Predicate) \wedge \\ argIs(nlPattern, 2, String)\end{gathered} \tag{4.63}$$

To be able to explain the simple Logic to NL engine, let's add the following assertions to our KB:

$$\begin{gathered} name(Coffee, \text{"coffee"}) \wedge \\ name(Restaurant1, \text{"L'Ardoise"}) \\ nlPattern(menuItem, \text{"\$1 has \$2 on the menu"}) \end{gathered} \tag{4.64}$$

Now, with this NL knowledge, our NL generation engine, when represented by the sentence

$$menuItem(Restaurant1, Coffee) \tag{4.65}$$

checks into the KB, whether it has enough of NL knowledge to do it. First it tries to find all the patterns of the predicate by issuing a query:

$$Q : nlPattern(menuItem, x) \tag{4.66}$$

which results in all the possible values of x:

Table 4.3: Results of query 4.66.

x
"$1 has $2 on the menu"

Then if does similar search queries for both of the concepts in the arguments of the predicate:

$$\begin{gathered} Q : name(Restaurant1, x_1) \vee namePlural(Restaurant1, y_1) \\ name(Coffee, x_2) \vee namePlural(Coffee, y_2) \end{gathered} \tag{4.67}$$

which results in

Table 4.4: Results of query 4.67.

x_1	y_1	x_2	y_2
"L'Ardoise"		"coffee"	

As we see in the result tables 4.3 and 4.4, it os able to find the pattern and also names of both of the arguments ($1 = $ *Restaurant*1 and $2 = $ *Coffee*). These results are merged into the NL representation "L'Ardoise has coffee on the menu.". The engine checks the *arity* of the *menuItem* predicate, , and verifies whether all of the slots are bound to non variable concepts. When this is fulfilled like in the example above, the engine knows that this sentence must be declarative (up to now we only have one declarative pattern), picks the proper pattern, and also adds the punctuation "." on it's own.

Now, consider, our Logic to NL engine gets the task to convert the formula (query)

$$menuItem(Restaurant1, x) \tag{4.68}$$

where x represents a variable as in other our examples. Our engine again, does the same query 4.66, but because the second argument is now a variable, only the first part (*Restaurant*) of the second query 4.67. Results for x_1 are the same as before, but there is no results for x_2. Now the engine detects that one of the pattern arguments ($2) is missing, and since it noes not have any additional info on how to convert a statement into an interrogative mode, it simply converts it into a KA pattern: "L'Ardoise has ___ on the menu.".

While these type of patterns is enough to be able to do the simple KA, it doesn't yet completely follow the examples from Table 4.1. To be able to do properly formed interrogative sentences when necessary, we need to add additional assertion, similar as the last part of assertion 4.64.

$$nlPattern(menuItem, \text{"What is on the menu in \$1"}) \tag{4.69}$$

Now the the query 4.66, if issued again will return more results:

Table 4.5: Results of query 4.66, after we have more *nlPattern* assertions for the predicate *menuItem*.

x
"$1 has $2 on the menu"
"What is on the menu in $1"

Now, considering that we are still converting $menuItem(Restaurant1, x)$, the engine will among two options pick a second one, because it has a better match against given arguments (only argument 1 is present), and thus instead of "L'Ardoise has ___ on the menu." convert into "What is on the menu in L'Ardoise?"

Careful reader might notice that the results for plural versions of concept names (*namePlural*) were always empty. For the concept $Coffee$, the plural version is exactly the same as for singular, so we omit it. In cases when this is not so, the engine picks the plural version when it is describing instances of *Class*, representing whole classes of terms which can have multiple instances, and singular version when describing instances of other terms.

The definitions (4.26, 4.24 and 4.25), are simplified formulations of minimal required NL knowledge to be able to present our approach. While the implementation of NL generation engine is totally up to the developer of such a system, for this examples (and also our implementation - see section 5.1.3), we decided to put the NL patterns into the KB, next to the definitions of the concepts. On the contrary to having the NL patterns in some external database or a text file, this approach in long term allows the system to ask questions about the language patterns themselves, and thus allow it to learn and ask about these as well. For example, if instead of strings directly, NL patterns would be constructed from the concepts of words (not part of our KB and not consistent with it, but serves as an hypothetical example, and more tightly matches our non-simplified implementation):

$$is(WordForPizza, Word) \wedge name(WordForPizza, "pizza") \wedge \\ nameWord(Pizza, WordForPizza)$$

then the system could notice that its missing the plural version of the word for the concept *Pizza* and ask for it: $namePlural(WordForPizza, x)$ (converted to NL: "What is the plural word for 'pizza'?".

Now we can define some more NL patterns, to be able to fully cover the examples from Table 4.1. Starting with *userLocation*:

$$nlPattern(userLocation, \text{"[\$2] Where are you?"}) \wedge \\ nlPattern(userLocation, \text{"Are you at \$2?"} \tag{4.70}$$

Which should be enough to produce conversions required for 1 from Table 4.1. The argument variable enclosed in "[]" means that this pattern should have that argument unbound, but it will not be used when generating the NL. Additionally, careful reader can notice

that these two NL patterns are fixed to a personal pronoun "you". We did this here in the approach for simplicity, but the implementation handles the pronouns properly (see subsection 5.1.3). Also, the functions generating NL can be given the addressee, so the system knows when to use person's name, "you" or even I, when talking about itself.

Then *is* predicate NL pattern:

$$\begin{gathered} nlPattern(is, \text{"[\$2] What kind of thing is \$1?"}) \wedge \\ nlPattern(is, \text{"\$1 is a \$2."}) \end{gathered} \tag{4.71}$$

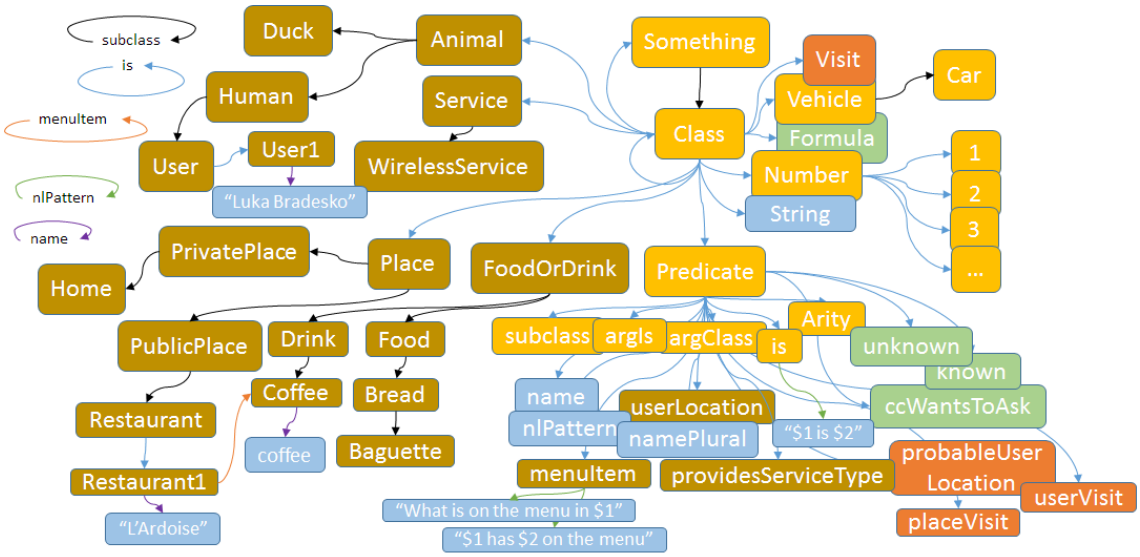

Figure 4.9: Current "NL generation knowledge" on top of the existing KB (new additions mareked with blue)

Also the pattern for *providesServiceType*:

$$\begin{gathered} nlPattern(providesServiceType, \text{"Does \$1 have \$2"}) \wedge \\ nlPattern(providesServiceType, \text{"Does \$1 provide \$2"}) \end{gathered} \tag{4.72}$$

And for *userOrdered*:

$$\begin{gathered} nlPattern(userOrdered, \text{"[\$3] What did you order?"}) \wedge \\ nlPattern(userOrdered, \text{"[\$3] What did you order as part of your visit \$2?"}) \end{gathered} \tag{4.73}$$

And the classes:

$$\begin{gathered} name(Restaurant, \text{restaurant}) \wedge \\ name(WirelessService, \text{"Wifi"}) \wedge \\ name(WirelessService, \text{"Wi-Fi"}) \wedge \\ name(Pizza, \text{"pizza"}) \wedge \\ namePlural(Pizza, \text{"pizzas"}) \wedge \\ name(Car, \text{"car"}) \wedge \\ name(FoodOrDrink, \text{"food or drink"}) \end{gathered} \tag{4.74}$$

4.4.2 NL to Logic

While simple processing based on the same NL knowledge as entered and described in the previous section (subsection 4.4.1) suffices to interpret isolated terms and denotational phrases from the user responses, converting general NL expressions back to logic is much trickier than the other way around, because natural language is much more ambiguous than logic. This process exceeds the scope of this KA related thesis and is described in more detail in the implementation (section 5.1.3), and paper (Schneider et al. 2015) and to other aspects of it also in chat-bot related works(R. Wallace 2013; Wilcox 2011). Following the simplistic examples as before just to show and be able to explain the approach, we explain the basic functionality (enough to be able to make our examples from Table 4.1).

In the proposed NL engine (described also in section 4.4.1 above), all of the *String* arguments of the predicates *name*, *namePlural* and *nlPattern* are indexed and searchable by the system, where the $# slots of the *nlPattern* strings are replaced with asterisks ($*$), representing any number of words, in a similar way as chat-bot scripts are matching defining their patterns(Wilcox 2011).

For illustration consider inverting the example 4.65 from the subsection 4.4.1. The system is presented with the statement

"Joe's Pizza has coffee on the menu." (4.75)

Because all the textual representations of the concepts are indexed, the NL engine is able to search for the strings appearing in the sentence through our NL predicate assertions and finds the following concepts:

Table 4.6: String search results for query 4.75.

String	**Concept**
"Joe's Pizza"	$Restaurant1$
"* has * on the menu"	$menuItem$
"coffee"	$Coffee$

Because of the *is* and *subclass* knowledge of $Restaurant1$ and $Coffee$ classes, which can be observed in Figure 4.9, and the $argIs$ and $argClass$ argument constraints of $menuItem$ (Definition 4.9), the inference engine is actually able to use the results above (Table 4.6), to construct the logical equivalent of the above sentence:

$$menuItem(Restaurant1, Pizza)$$

This is done in the following sequence:

1. The system first finds argument constraints of all the predicates that are found in the results. In this case it is $argIs(menuItem, 1, Restaurant)$ and $argClass(menuItem, 2, FoodOrDrink)$ for the predicate $menuItem$.

2. All of the predicates are checked for their arity, and removed if the arity of the predicate is not the same as the number of concepts that were found as part of the sentence. In this case the arity is 2 (Assertion 4.36), and we found 2 unique concepts ($Coffee$ and $Restaurant1$).

3. The concepts that are not filtered out in the previous steps are then checked whether they fulfill the the arity constraints of the arguments of the predicates. In our above example, $Coffee$ is a subclass of $FoodOrDrink$, so it can be used as argument 2, and $Restaurant1$ is an instance of $Restaurant$, so it can be used as argument 1.

To make the example more complicated, and thus process more clear, consider an additional predicate *servesCuisine* which has a totally different meaning than *menuItem*, but can have the same language pattern:

$$is(servesCuisine, Predicate) \wedge$$
$$arity(servesCuisine, 2) \wedge$$
$$argIs(servesCuisine, 1, Restaurant) \wedge$$
$$argClass(servesCuisine, 2, Cuisine) \wedge$$
$$nlPattern(servesCuisine, \text{"\$1 serves \$2"})$$

while we add one more pattern to the *menuItem* predicate.

$$nlPattern(menuItem, \text{"\$1 serves \$2"})$$

With this additional predicate and NL knowledge, the engine, given the following text:

"Joe's Pizza serves coffee." (4.76)

will find the following possible terms:

Table 4.7: String search results for query 4.76.

String	**Concept**
"Joe's Pizza"	$Restaurant1$
"* serves *"	$menuItem$
"* serves *"	$servesCuisine$
"coffee"	$Coffee$

Now on the Table 4.7 we can see, that there are two predicates for the same NL pattern. But when the system will try to fit the class $Coffee$ into both of them, it will not fit to *servesCuisine*, because it requires subclass of *Cuisine*, but $Coffee$ is a sub-class of *Drink*. Consequently, the predicate *servesCuisine* will be filtered out, and the system will properly disambiguate into the logical equivalent

$$menuItem(Restaurant1, Coffee)$$

These two examples illustrate a simplest kind of sentential NL conversion. When there are many more possible concepts which need to be combined, the complexity of the problem quickly explodes, or is ambiguous even given the constraints. For this reason, our actual Cyc SCG implementation contains much more complex KB structures and patterns which help with the conversion and a high-speed parser.

For the most complicated NL conversions, similar approach to a standard ChatScript or AIML patterns(Wilcox 2011; R. Wallace 2013) is taken, just that the patterns are not matched to textual response patterns directly, but to the logical statements instead. Consider the example answer given as step 5 in (**tab:conversation1**), when user would, instead of simple "pizzas", answer something like "They sell pizzas", or "They have pizzas on the menu". In this case the simple approach described at the beginning of this chapter would not find anything and thus the system needs more descriptive patterns like

$$nlPattern(menuItem, \text{"\$1 (sell|sells|has|have) \$2 |(on the menu)"}) \quad (4.77)$$

where the sign | represent the or clause, meaning that the "on the menu" is optional and in the middle, only one of the words ("sell","sells",...) will be used. Additionally, the

system needs to infer that "They" refers to the restaurant where user currently is (first argument), and that "Pizzas" refer to the the second argument. Additional help that the NL system has and we didn't mention is the predicate we used to generate a question. For example, in the case of step 5, the system can give advantage to *menuItem* predicate and its argument constraints, even if there are multiple ambiguous predicates or other terms matched by the strings.

Also, note that these more complicated patterns are to be used only when the assisted KA does not present valid options and user types free text and also, that the patterns do not ever specify what should be the systems response (as opposed to how chat bots work). Instead the pattern just converts the text to the logic and then leaves up to the inference engine what to do next. This is completely different approach from standard chat-bot systems where patterns dictate the responses as well. For readers interested in more detailed overview of chat-bot approaches and patterns please refer to chapter Related Work (sub-chapters 3.2.6 and 3.2.7), or related paper (Bradeško and Mladenić 2012).

4.4.3 Dialog Formulation

As mentioned before, *Curios Cat* conversation is not predefined by the patterns, but is completely knowledge driven. The knowledge is being inserted into the system by its users (section 4.5), and automated context (section 4.3). This is then picked up by the inference engine that generates questions and comments with the help of KA rules (section 4.2.3). The resulting assertions, which propose the intents like $ccWantsToAsk(x, y)$ (definition 4.17) and $ccWantsToComment(x, y)$ (definition 4.27 below), are then converted to the natural language and shown to the users, which then respond. The appearance of the assertions and consequently questions/comments in NL and their user responses form a conversation. The topic and flow of the conversation in the most simplistic form depends on the order that these assertions appear in the KB (the newest ones are always displayed first).

To be able to produce commenting or advice statements, beside the questions (as shown also in the examples in Table 4.1), we are missing one more predicate.

Definition 4.27 (predicate "*ccWantsToComment*"). This conversational predicate written as $ccWantsToComment(x, y)$, denotes that the *Curious Cat* system wants to tell (or comment) a statement y to the user x. For example, assertion

$$ccWantsToComment(User1, menuItem(Restaurant1, Coffee)$$

tells CC system to tell user $menuItem(Restaurant1, Coffee)$, which, after it goes through logic to NL conversion (section 4.4.1) is paraphrased as "L'Ardoise has coffee on the menu.". The predicate is formally defined as follows:

$$\begin{gathered} is(ccWantsToComment, Predicate) \wedge \\ arity(ccWantsToComment, 2) \wedge \\ argIs(ccWantsToComment, 1, User) \wedge \\ argIs(ccWantsToComment, 2, Formula) \end{gathered} \tag{4.78}$$

For the experiments produced in this paper, *Curious Cat* system (Procedural Component) tracks additional context of users current concept (topic) of interest (it asserts it into KB with *currentTopic* predicate), and also a list of newly created concepts by that user. For example, concepts *JoesPizza* and *PizzaDeluxe* from our conversation example are on this ordered list. In cases it has multiple *ccWantsTo...* formulas in place, it prefers the formulas which contain these concepts in the given order. Then it simply exploits the

feature of the system to produce new questions/comments on the fly as all users (including the one having the conversation) and contextual data are constantly producing new knowledge and consequently new questions/comments. It then presents these questions/-comments to the user as they come in. For example, when user answers that he is at Joe's Pizza restaurant, the system produces $ccWantsToAsk(User1, menuItem(JoesPizza, x))$, which is showed to the user in NL, as it is the newest formula that appeared and contains the concept of $JoesPizza$. This simple approach works quite well, since the latest formula is always a consequence of the previous user action. Additionally, this can be used for pro-activity, when the question/comment for the user appears as a consequence of some other user's action, or a context change. If the user did not use the system for a while, we can simply present him with the latest formula when it appears from the inference as a proactive question/comment. In the cases when we have a statement, and a question formula, the system presents the comment and then a new question as part of the same response (as also visible in example conversation in Table 4.1).

For the cases when the system needs more explicit control over the order of the questions, the inference rules can be added using special intent predicates which allow ordering of the responses. For example predicates like $ccWantsTo...$, but with arity of 3, where new argument is importance, and then the system picks more important predicates first. Additionally, the system tracks the window of user and its own responses with logical assertions like $ccResponse(user, formula, num)$ and $userResponse(user, formula, num)$ which stores the history of the conversation and allows rules like:

$$\forall x \forall y_1 \forall y_2 : userResponse(x, y_1, 0) \wedge userResponse(x, y_2, 1) \wedge equals(y_1, y_2) \implies ccWantsToComment(x, userRepeatingItself(x))$$

This (purely) example rule will produce a comment that user is repeating himself, when the user will respond with the same thing two times in a row.

At last, in the occasions when there are no new questions appearing on the list, the system will pick from the old ones in the order, until something will trigger a new one, or until depleting all of them. If the depletion happens, then the topic will traverse all the concepts from the history and thus accidentally trigger additional inference with new topic of interest for the user, which will produce new questions, which will additionally (if answered) produce new follow-up questions/comments. As the final fullback, the system will randomly pick a concept from its KB and present it to the inference engine as a current interest for the user.

Now we can add some dialog formulation rules to be able to produce the examples from Table 4.1. First, we want to define that the engine asks the user where he is, when it doesn't know that:

$$\forall x \exists y : unknown(userLocation(x, y)) \implies ccWantsToAsk(x, userLocation(x, \$z) \tag{4.79}$$

This rule basically detects when there is an user in the KB, which doesn't have the $userLocation$ assertion and generates a question "Where are you(user x)?".

Additionally, when the *Procedural Component* detects probable location, this should trigger a "confirm" question:

$$\forall x \exists y \forall z : unknown(userLocation(x, y)) \wedge probableUserLocation(x, z) \implies ccWantsToAsk(x, userLocation(x, z) \tag{4.80}$$

This rule detects the insertion of *probableUserLocation*, and if it's not yet confirmed, it causes the inference to produce a yes/no question (when all the variables in the question are bound to something, user can only confirm or reject the proposed question. There is nothing to be filled in): "Are you at Z?".

Now lets define the rest of rules that are used in our examples from Table 4.1. and KA dialog formulation.

To be able to ask about the Wi-fi signal, we need a vocabulary for storing this information and also for asking. This is done by introducing the rule which generates the questions about the *providesServiceType* predicate which was defined in the assertion 4.38. The rule triggers when user is at the public place (not any place, to prevent too many questions):

$$\begin{gathered}\forall x \forall u : userLocation(u, x) \wedge is(x, PublicPlace) \wedge \\ unknown(providesServiceType(x, WirelessService)) \\ \Longrightarrow \\ ccWantsToAsk(providesServiceType(x, WirelessService))\end{gathered} \tag{4.81}$$

Definition 4.28 (Predicate $hasGoodWifi$). To be able to ask additional question about Wi-fi signal, introduce $hasGoodWifi$ predicate, formally defined as follows:

$$\begin{gathered}is(hasGoodWifi, Predicate) \wedge \\ arity(hasGoodWifi, 1) \wedge \\ argIs(hasGoodWifi, 1, PublicPlace) \wedge \\ nlPattern(hasGoodWifi, \text{"Is it fast enought to make Skype calls"}) \wedge \\ nlPattern(hasGoodWifi, \text{"Is Wifi at \$1 fast enough to make Skype calls"})\end{gathered} \tag{4.82}$$

Then, the KA rule that triggers follow-up questions once we know that there is a wi-fi signal:

$$\begin{gathered}\forall x \forall u : userLocation(u, x) \wedge is(x, PublicPlace) \wedge \\ providesServiceType(x, WirelessSignal) \wedge \\ unknown(hasGoodWifi(x) \\ \Longrightarrow \\ ccWantsToAsk(hasGoodWifi(x))\end{gathered} \tag{4.83}$$

On top of just follow-up questions, we can define follow-up conversation which is also time-based (using time of visit info as defined in section 4.3 and example assertion 4.57). For this we rely on the *greaterThan* predicate and *userVisit* assertions, to write rules like the following:

$$\begin{gathered}\forall u \forall v \exists t \forall d \forall x : userVisit(u, v, t, d) \wedge \\ greaterThan(d, 600) \wedge \\ unknown(userOrdered(u, v, x) \\ \Longrightarrow \\ ccWantsToAsk(u, userOrdered(u, v, \$x))\end{gathered} \tag{4.84}$$

The rule above triggers once the Context part of *Procedural Module* asserts that the user was at some place for more than 10 minutes, and produces the question as given in the example step 6 in Table 4.1.

4.5 Consistency Check and KB Placement

As already described, we can employ the inference engine to deduce various facts using forward-chaining inference including intents to ask a question. The answers can then be automatically inserted into the KB. The knowledge stored in the KB can be then retrieved using logical queries, which return the matching knowledge, or it can infer additional entailed knowledge at query time using backward-chaining inference. The queries are simple logical formulas, which the inference tries to prove or satisfy. For example, the query

$$Q : class(x, Food) \tag{4.85}$$

will return the following results if queried over our current KB (Figure 4.9)

Table 4.8: Results for query 4.85.

x
Bread
Baguette
Pizza
Margherita

Or the query

$$is(Restaurant1, Place) \tag{4.86}$$

will return *True*.

For illustration, let us assume that the system finds the following question to ask

"What has Joe's Pizza on the menu?"

$$(menuItem(JoesPizza, x))$$

with the argument constraints on the predicate as defined in assertion 4.36. Assuming that the user answers with "Pizza Margherita" (hypothetical example, different than in Table 4.1, step 12) which exists in our KB, the system can ask the inference engine the following type of general query:

$$\begin{gathered} Q : name(x, \$answer) \wedge \\ argIs(\$pred, \$argPos, y) \wedge \\ is(x, y) \wedge \\ \left(\begin{array}{c} argClass(\$pred, \$argPos, z) \wedge subclass(x, z) \\ \vee \\ unknown(\exists z : argClass(\$pred, \$argPos, z)) \end{array} \right) \end{gathered} \tag{4.87}$$

where we replace the meta variables (marked with \$) with our values from the example question, where $\$answer$ represents user answer, $\$pred$ represents the predicate that was used to propose a question, and $\$argPos$ is the position of the variable in the question query. After we fill in our values ($\$answer$ = "Pizza Margherita, $\$pred = menuItem$, $\$argPos = 2$), we get the following concrete logical query:

$$\begin{gathered} Q : name(x, \text{"Pizza Margherita"}) \wedge \\ argIs(menuItem, 2, y) \wedge \\ is(x, y) \wedge \\ \left(\begin{array}{c} argClass(menuItem, 2, z) \wedge subclass(x, z) \\ \vee \\ unknown(\exists z : argClass(menuItem, 2, z)) \end{array} \right) \end{gathered} \tag{4.88}$$

If this query is asked in our current KB (Figure 4.9), we get the following results:

Table 4.9: Results for query 4.88.

x	y	z
Margherita	*Class*	*FoodOrDrink*

Because the inference engine was able to answer this query, we immediately know that the answer is consistent with the KB and is safe to assert it as $menuItem(JoesPizza, Margherita)$.

At this point we have seen an example of how to add new knowledge when the concept that the user has provided already exists in the KB and the answer is structurally valid. But what would happen if user should say "Pizza Deluxe" (as in our example in Table 4.1, step), which we do not have in the KB. In this case the query above would return nothing. This would happen as well, if the user should say "car" (as in the step 6). When the validation query does not return results (as would happen for "car"), we need to separately check whether the concept exists:

$$Q : name(x, \text{"car"}) \tag{4.89}$$

If it does (i.e. the query 4.89 above returns the resulting concepts), then the answer is actually invalid on structural grounds (the term it includes ca not be used a viable answer). But, if the concept does not exist yet (nothing returned, as would be the case for "Pizza Deluxe"), then we can simply create it together with its NL denotation and assert it using our question predicate.

4.5.1 Detailed Placement in the KB

In addition to deciding to add some newly acquired knowledge to the KB, the exact location in the "KB graph" is determined by the *argIs* and *argClass* assertions on the question predicate. Consider our example question $menuItem(JoesPizza, x)$. Because of the $argClass(menuItem, 2, FoodOrDrink)$, the answer must be a subclass of $FoodOrDrink$ concept. Let us say the answer is "Pizza Deluxe", as with previous explanations. To insert the knowledge, the system goes through the steps described in the previous section 4.5. Because of *argClass* assertions $PizzaDeluxe$ concept is added as a *subclass* of $FoodOrDrink$ as illustrated in Figure 4.10.

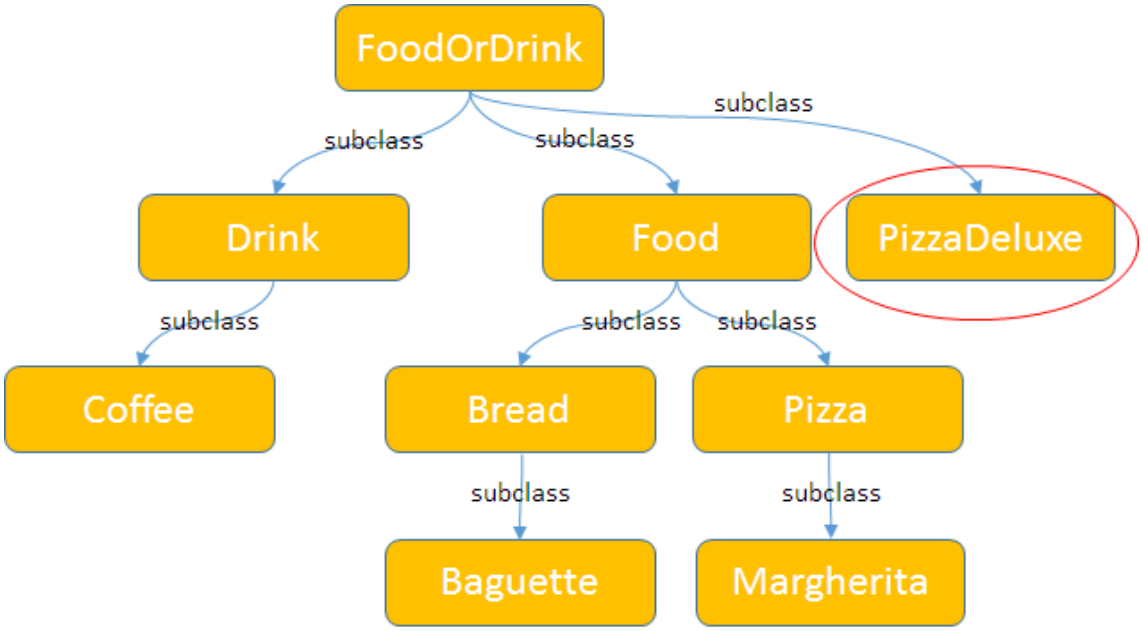

Figure 4.10: *FoodOrDrink* part of the class hierarchy from our example KB with newly added concept.

While this is logically valid, it is not detailed enough to satisfy our KA requirements. For this reason, when the concept does not already exist, or is not detailed enough (too high in the class hierarchy), we can issue additional query:

$$Q : subclass(x, FoodOrDrink) \tag{4.90}$$

which, for our example knowledge it returns:

Table 4.10: Results for query 4.90.

x
$FoodOrDrink$
$Drink$
$Coffee$
$Food$
$Bread$
$Baguette$
$Pizza$
$Margherita$

This gives us various options.

1. Ask the user, which one of these *PizzaDeluxe* is (excluding the main class $FoodOrDrink$). For example: "What describes it in most detail?", or: "Is Pizza Deluxe a drink, coffee, bread, baguette, pizza or Margherita"

2. Ask for the first level subclasses first, then for the next level, etc.: 1. "Is Pizza Deluxe a type of drink or food?", 2. "Is Pizza Deluxe a type of pizza?"

3. (which is usually the best option) we can scan all of the resulting concepts (results from Table 4.10) for their NL patterns and names, and then match the strings to "Pizza Deluxe", to see which one fits the best. In this case, only the *Pizza* concept provides a partial match ("Pizza" vs "Pizza Deluxe"). So, we can immediately ask: "Is Pizza Deluxe a type of Pizza?". If the user agrees, we can, in addition to $subclass(PizzaDeluxe, FoodOrDrink)$, assert: $subclass(PizzaDeluxe, Pizza)$. This results in our new concept being located in the KB at the most descriptive place as illustrated in Figure 4.11.

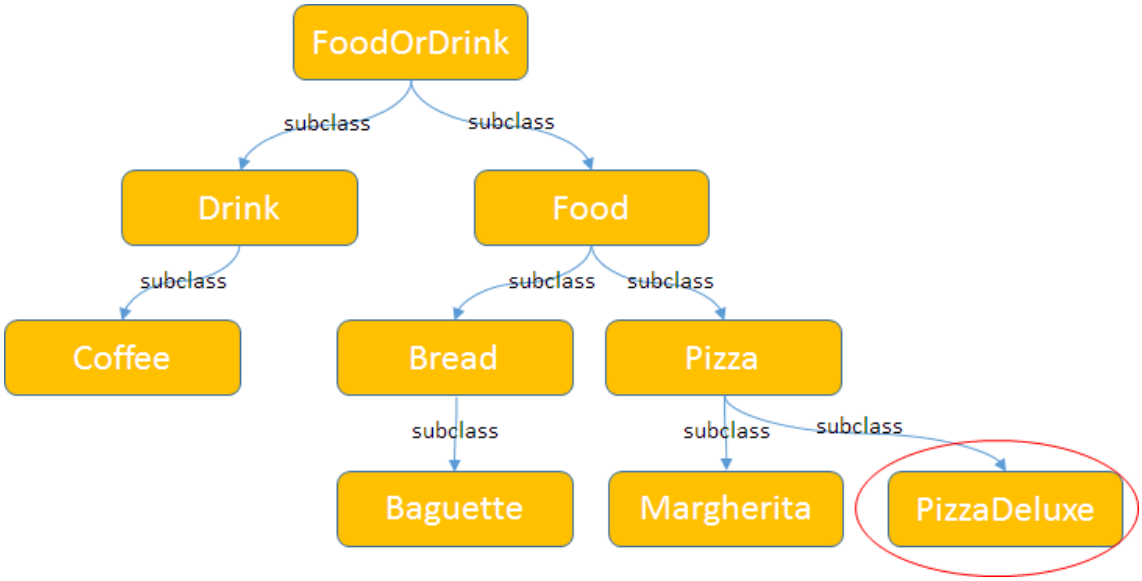

Figure 4.11: New position of the *PizzaDeluxe* concept in our KB.

For the cases, when this more detailed placement is not done immediately after the first entrance of the concept, and cannot be handled by the *Procedural Component*, we can have a special rule which will cause the system to ask questions about more detailed positions:

$$\begin{gathered} \forall u \forall x \exists y \forall z : is(u, User) \wedge \\ subclass(x, y) \wedge \\ subclass(x, z) \wedge \\ \neg equals(z, x) \wedge \\ unknown(\neg subclass(z, x)) \\ \Longrightarrow \\ ccWantsToAsk(u, subclass(z, x)) \end{gathered} \tag{4.91}$$

Or the same for instances:

$$\begin{gathered}\forall u \forall x \exists y \forall z : is(u, User) \wedge \\ is(x, y) \wedge \\ subclass(y, z) \wedge \\ unknown(is(x, z)) \wedge \\ unknown(\neg is(x, z)) \\ \Longrightarrow \\ ccWantsToAsk(u, is(x, z))\end{gathered} \tag{4.92}$$

The rule above checks for each instance of some class, what are subclasses of that class, and then checks whether the concept can be an instance of any of these (if not already, or strictly asserted that not), then is proposes a list of questions for every of the possibilities. For example, $ccWantsToAsk(User1, is(JoesPizza, Restaurant))$ which converts to "Is Joe's pizza a restaurant?". This rule works in cooperation with *class* transitivity rule (4.10), and *is* transfer rule (4.24). If we do a slight change in the rule consequent ($ccWantsToAsk(u, is(x, \$z))$, this rule generates open-ended questions like "What kind of thing is Joe's pizza?" In our implementation (chapter 5), this was done with a slightly more complicated approach (using custom predicate), which allowed us to include the initial class name in the questions as well ("What kind of **place** is Joe's pizza?").

4.6 Crowdsourcing Mechanisms

Up to this point, we have discussed the mechanisms of KA, showing how it is possible to get valid knowledge from a single user. In order to lower the cost of and/or increase the speed of KA we can involve a crowd. Crowdsourcing however bring several challenges (as described in Section section 2.5) including the following:

- User privacy
- Users making deliberately false claims or having mistaken ideas about the world
- Fast changing state of the real world

We tackle this by organizing our KB into smaller, hierarchical knowledge base structures. Each of these structures is then our virtual knowledge base in which we operate, and which has its own independent knowledge, added on top of all the sub-KBs up in the hierarchy (Figure 4.12). In the Cyc system, these contextual KB structures are called *Microtheories*(Johan de Kleer 2013).

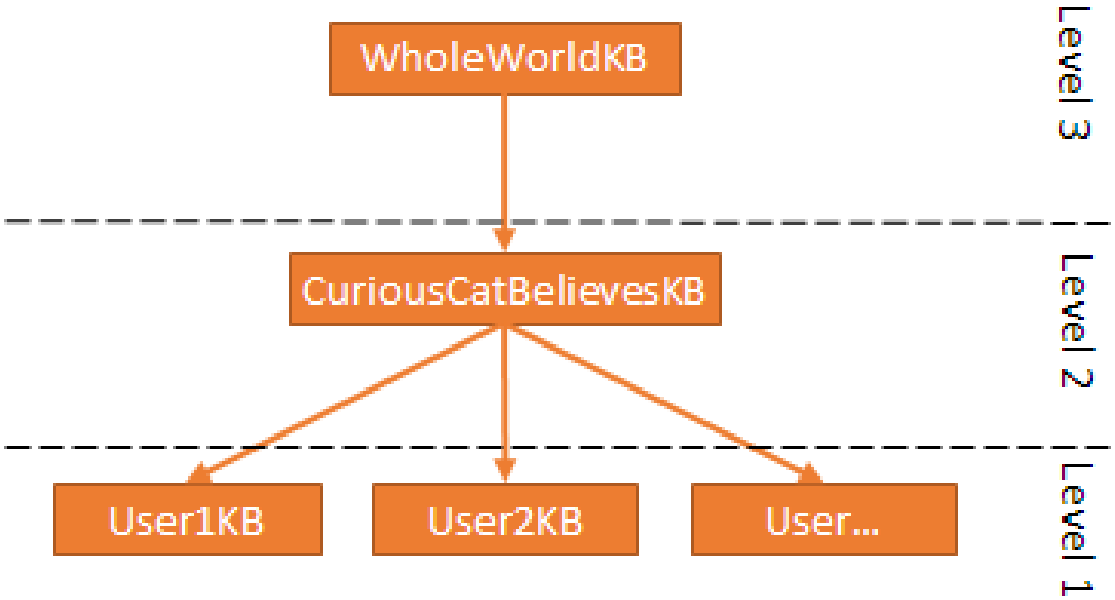

Figure 4.12: Hierarchical structure of knowledge bases.

The idea is that the sub-KBs which are higher up in the hierarchy are not aware of the knowledge that is at the lower levels. But the KBs which are at the lower levels, contain all the knowledge of their ancestors from the higher levels. For example, let us assume that $WholeWorldKB$ in Figure 4.12 has the knowledge defined in subsection 4.2.1 and subsection 4.2.2, $CuriousCatBelievesKB$ has the knowledge defined in section subsection 4.2.3 and then acquired from multiple users. $User1KB$ has knowledge that was acquired from $User1$ and $user2KB$ has knowledge acquired from $User2$ (new, other user). Following our KB structure given in Figure 4.7 and formal logical definitions, we can see that:

- $WholeWorldKB$ contains the upper ontology and preexisting knowledge from Figure 4.7.
- $CuriousCatBelievesKB$ contains KA assertions and union of assertions acquired from all the users.
- $User1KB$ contains assertions acquired from user 1 and only relevant to her/him.
- $User2KB$ contains assertions acquired from user 2 and only relevant to her/him.

Following this structure, it becomes obvious that each user in our system has its own sub-KB, connected to the main knowledge only through $CuriousCatBelievesKB$. Also, $User1KB$ and $User2KB$ cannot see each others knowledge, but only perceive the world through the "eyes" of $CuriousCatBelievesKB$ and their own local sub-KB. This means, that if $User1$ lies about something, the wrong knowledge will be only available to $User1$, while the rest of the users will not be affected. In this way, the users privacy is also protected.

To benefit from crowdsourcing we want to share some of the knowledge to all of the users. Since each sub-KB can "see" only the assertions stored in itself and the assertions

higher up in the hierarchy, we can control what only one user knows, versus all the users, by moving the specific assertions up or down through the KB hierarchy. We are proposing two approaches:

1. Crowdsourcing through repetition (subsection 4.6.1),
2. and 2) Crowdsourcing through voting (subsection 4.6.2).

Crowdsourcing approach 1 is more advanced, since it includes type 2 as well, once the repeated assertions from multiple users got promoted to $CuriousCatBelievesKB$. Because it only promotes the knowledge which is asserted by multiple users independently, there is reduced chance for temporal wrong knowledge staying in our system before the voting from approach 2 removes it. In this sense, Crowdsourcing through repetition gives better results, but it has a drawback, especially if there is not enough of users in the system, the knowledge takes longer to be promoted to the main KB where other users could benefit from it. Additionally, users might have a feeling that they are the only one in the system if they dont see some immediate feedback and activity. Initially we started with Crowdsourcing through repetition, but then decided (due to reasons above) to do our initial experiments (described in chapter 6, subsection 6.1.4) by using the "voting" option.

Additionally to store the answers of different users in different micro theories, the system uses the voting results of users to influence the system's KA behavior towards these particular users (It is used as part of acquired user context, subsection 4.3.2). For example, if user states that she agrees with some statement, this can be used to produce additional questions, or influence the questions that would appear otherwise. Similarly if the user doesn't agree, or doesn't know. This is especially useful to not ask users about particular concept, if they answered with "I don't know" at some point. For this, we use the following predicates that get asserted into the KB for each particular vote that appears in the system.

Definition 4.29 (Predicate $userAgrees$). To be able to state that user agrees with something, CC uses the predicate $userAgrees$, formally defined as follows:

$$\begin{gathered} is(userAgrees, Predicate) \wedge \\ arity(userAgrees, 2) \wedge \\ argIs(userAgrees, 1, User) \wedge \\ argIs(userAgrees, 2, Formula) \end{gathered} \tag{4.93}$$

Definition 4.30 (Predicate $userDoubts$). To be able to state that user does not agree with something, CC uses the predicate $userDoubts$, formally defined as follows:

$$\begin{gathered} is(userDoubts, Predicate) \wedge \\ arity(userDoubts, 2) \wedge \\ argIs(userDoubts, 1, User) \wedge \\ argIs(userDoubts, 2, Formula) \end{gathered} \tag{4.94}$$

Definition 4.31 (Predicate $userDontKnow$). To be able to state that user doesn't know the answer, CC uses the predicate $userDontKnow$, formally defined as follows:

$$\begin{gathered} is(userDontKnow, Predicate) \wedge \\ arity(userDontKnow, 2) \wedge \\ argIs(userDontKnow, 1, User) \wedge \\ argIs(userDontKnow, 2, Formula) \end{gathered} \tag{4.95}$$

4.6.1 Crowdsourcing through repetition

This type of proposed crowdsourcing mechanism is based on the number of identical assertions in all the sub-KBs on the same level (i.e., different users providing exactly the same knowledge). Once the specific assertion count is above a threshold, we assume that there is enough evidence for promoting it to general knowledge, which is performed by moving the knowledge to higher level of the hierarchy (via "lifting rules" in the $CuriousCatBelievesKB$), and thus making it visible for all the users. After the knowledge is in the public KB, crowd users can start voting on it (see section 4.6.2).

Consider the case of $User1$ answering the question from the previous examples: "What did you order?", with a lie: "spicy unicorn wings". The system will go through steps described in section 4.5 (Consistency Check and KB Placement), and if the user confirms this new concept, *Curious Cat* will believe (in the world for $User1$), that he ate spicy unicorn wings and that the $JoesPizza$ restaurant has them on the menu. The contextual KB $User1KB$ would then get the assertion $menuItem(JoesPizza, SpicyUnicornWings)$. There is a very low chance that any other user would provide the same answer, so the wrong assertion will never get promoted to higher levels of KB.

On the other hand, if $User1$ answers $menuItem(JoesPizza, PizzaDeluxe)$, then $User2$ answers the same, then $User3$, the assertion already has a count of more than 2, which is the threshold in our system, and the assertion will get promoted to be visible for all the users (world). For newcomers to $JoesPizza$, the system will already know that they have "Pizza Deluxe" on the menu. The real world implementation example of promoted knowledge at the first visit can be seen on Figure 4.13.

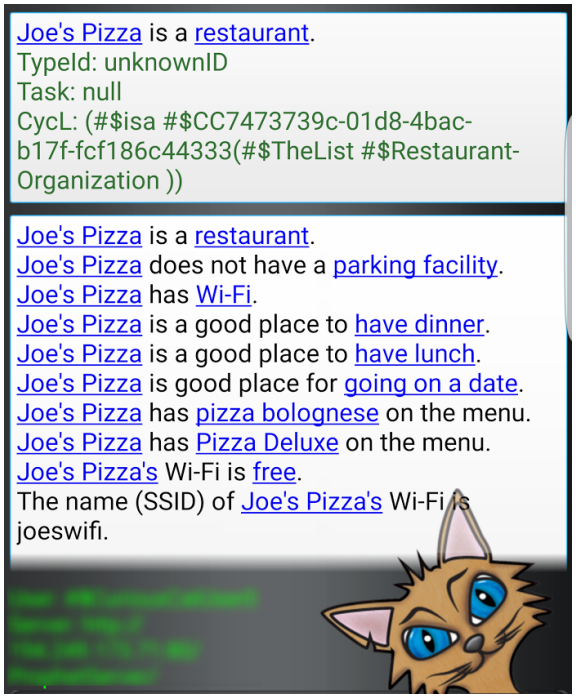

Figure 4.13: Screenshot of Curious Cat displaying an already public information about Joe's Pizza to a visiting user.

4.6.2 Crowdsourcing through voting

Contrary to the previous example, we can also promote all the answers immediately. In this case, all the assertions except the private ones (e.g. ones that contain the concept for the $User\#$), will get asserted into both the users KB and the $CuriousCatBelievesKB$.

Once the assertions are in $CuriousCatBelievesKB$, users will not get the standard questions produced by the KA rules, but will still get occasionally the "crowdsourcing" questions (produced by different rules), which will simply be checking the truth of some existing knowledge: "Is it true that Joe's Pizza has Pizza Deluxe on the menu?". These simple yes/no questions allow us to assess the truthfulness of the logical statement and remove it, if it gets more negative answers than positive.

This voting mechanism serves for promoting knowledge (as described in subsection 4.6.1 (Crowdsourcing through repetition) and for detecting when the world changes and something that was true in the past is not true anymore.

4.7 Putting it All Together

This section wraps up the approach and brings together the components described in chapter 4 into a coherent system, which we try to describe through the example conversation presented in Table 4.1 and Table 4.2. Each subsection below will explain the work-flow and mechanisms required to produce one of the steps.

4.7.1 Step 1, Where are you?

Let's say our example user $User1$ (defined with assertion 4.35, see 4.6) was traveling for a while and now arrived at some restaurant. He is there for 3 minutes and just managed to find a table and sit down.

He didn't touch his phone in a while, but the *Curious Cat* periodically wakes up and send a GPS coordinate to the *Procedural Component* on the server side. While he was traveling, the *SPD* algorithm (subsection 5.4.2) was only returning data about the route. Now, after the raw GPS coordinates were suddenly grouped around stable place, the algorithm detects the cluster and its center, calls the Foursquare API and retrieves the data about the location (restaurant called "Joe's Pizza).

Now the *Procedural Component* does the following steps:

1. Converts the name of the place into a constant name $JoesPizza$ and next available concept of a visit $Visit1$ (see assertions 4.54 and 4.55).
2. Asserts the visit time, and current duration in the KB (assertion 4.57)
3. Checks if it exists in our KB, and adds it if it doesn't (assertion 4.54).
4. It also creates the new concept for visit, $Visit1$ (4.55), and asserts the time of arrival and current duration of stay (4.57).
5. It removes all the existing occasions of $userLocation(User1, x)$ and $probableUserLocation(User1, x)$
6. It adds a new assertion $probableUserLocation(User1, JoesPizza)$ (see assertion 4.56).

This last step triggers the inference engine (rules 4.79 and 4.80), to generate $ccWantsToAsk$ questions:

- $userLocation(User1, x)$

- $userLocation(User1, JoesPizza)$

Because these two are the same predicate, and the second one is better defined, the system merges them into a single response, where the main question is the second assertion (better defined).

These assertions are then converted to NL (according to assertions in 4.70), and presented to the user as 1 (Table 4.1). Because the main question has no variables, it can be only true or not, so *CC* already provides the "yes/no" options as part of the possible answer.

Once user answered, the assertion $userLocation(User1, JoesPizza)$ got asserted into the KB, and $JoesPizza$ set as the current topic. This also caused inference engine to retract the questions, since user location is not unknown anymore. At the same time the *Procedural Component* asserts that the place that user visited is actually the $JoesPizza$, as already shown in assertion 4.59.

4.7.2 Step 2. What kind of place this is?

After the $userLocation(User1, JoesPizza)$ got asserted into the KB, the inference engine runs through all the rules that use this predicate in the antecedent, which causes the rule 4.92 (the version with unbound variable) to assert $ccWantsToAsk(User1, is(JoesPizza, x))$. This according to assertion 4.71 produces the NL text "What kind of thing is Joe's Pizza?" (step 2, Table 4.1).

When user answers with "Restaurant", the engine (as described in section 4.5), replaces the general query 4.87 with the appropriate instance:

$$\begin{array}{c} Q : name(x, "restaurant") \wedge \\ argIs(is, 2, y) \wedge \\ is(x, y) \wedge \\ \left(\begin{array}{c} argClass(is, 2, z) \wedge subclass(x, z) \\ \vee \\ unknown(\exists z : argClass(is, 2, z)) \end{array} \right) \end{array} \tag{4.96}$$

which returns the results:

Table 4.11: Results for query 4.96.

x	y	z
Restaurant	*Class*	/

which is a signal to the system (as described in section 4.5), that this answer is consistent with our KB and can be asserted ($is(JoesPizza, Restaurant)$).

4.7.3 Step 3. Does Joe's Pizza have Wi-Fi?

The assertion ($is(JoesPizza, Restaurant)$) which got asserted as a consequence to answering step 2 from the Table 4.1 (see previous subsection above), triggers the rule 4.81, which produces the following assertion:

$$ccWantsToAsk(providesServiceType(JoesPizza, WirelessService)).$$

Then, it gets converted into NL ("Does Joe's Pizza have Wi-Fi?", assertion 4.72), and once user responds with "yes", the assertion ($providesServiceType(JoesPizza, WirelessService)$) goes into the KB under the User1's micro-theory.

4.7.4 Step 4. Is it fast enough to make Skype calls?

Similarly as previous question, this one is triggered after the system knows Joe's Pizza has Wi-Fi (due to $\wedge$ constraint in the antecedent of rule 4.83). This rule produces the $ccWantsToAsk(hasGoodWifi(JoesPizza)$, which converted to NL appears as "Is it fast enough to make Skype calls?", as defined in assertion 4.82 and visible in example step 4 in Table 4.1. This question is presented to the user before other possible questions that are there before (such as one from step 5, because it appeared directly after answering something, it has the current topic's concept in its logical representation and is thus a question that appear to be most up-to date regarding the current conversation.

After user answers with "I don't know" (Table 4.1), this is picked up by the *Procedural Component* and asserted into the KB as $userDontKnow(User1, hasGoodWifi(JoesPizza))$, which can then be used by further rules.

4.7.5 Step 5. What's on the menu at Joe's Pizza?

When the user answered that the place is a restaurant, simultaneously with already described rules and questions, also the rule 4.47 was triggered (see additional explanatory assertion 4.48 and 4.7). This rule triggered because of the existing knowledge about $Restaurant1$, which includes $menuItem(Restaurant1, Coffee)$, so the engine figured out that also $JoesPizza$ can have something on the menu. This rule produces the

$$ccWantsToAsk(User1, menuItem(JoesPizza, x))$$

assertion, which converts to "What's on the menu at Joe's Pizza?" (due to the assertion 4.69) in NL. When user answers with "pizzas", the concept $Pizza$ is found by the system (due to the assertion 4.74), and the assertion $menuItem(JoesPizza, Pizza)$ is entered into the kb.

4.7.6 Step 6. What did you order?

As was described in the sections 4.3 and 5.4.2, *Procedural Component* and the *CC* client are waking up periodically to mine the user context and among other things, also assert how long the user is at a particular location. In this case (following our example conversation), the place is Joe's Pizza, represented by $JoesPizza$ and $Visit1$ concepts. Consequently, even when user is not using *CC*, the system is updating the assertion $userVisit(User1, Visit1, 1505746857, t)$, to reflect the time of stay at particular location (represented by t, see assertions 4.57 and 4.58). Once the user stays at the place for more than 10 minutes (600 seconds), the rule 4.84 will fire and trigger the *CC* to ask $ccWantsToAsk(User1, userordered(User1, Visit1, x))$ question, which converted to NL (assertion 4.73) appears as "What did you order?". After the user answers with a "car" (maybe trying to test the system), *CC* responds as described in the next subsection (subsection 4.7.7) below.

4.7.7 Step 7. I've never heard of food or drink called 'car' before.

After the user answers with a "car" to the question in step 6, the *consistency check* subsystem of *CC* checks the argument constraints of the predicate *userOrdered* via the query defined in the assertion 4.87. First it modifies this query to reflect the proper predicate:

$$\begin{array}{c} Q : name(x, \text{"car"}) \wedge \\ argIs(userOrdered, 3, y) \wedge \\ is(x, y) \wedge \\ \left(\begin{array}{c} argClass(userOrdered, 3, z) \wedge subclass(x, z) \\ \vee \\ unknown(\exists z : argClass(userOrdered, 3, z)) \end{array} \right) \end{array} \tag{4.97}$$

The query (query 4.97) above doesn't return any results, so according to the approach described in section 4.5, the system goes to check whether it knows anything about the concept called 'car', as shown in the query 4.89 ($Q : name(x, \text{"car"})$. Since this query returns the result (according to our existing KB) Car, CC KA Engine can know that this answer is actually inconsistent, because the previous query (query 4.97) would otherwise return the resulting concept. For this reason the *Procedural Component* responds with a double check question, making sure that the user really meant the answer: "I've never heard of food or drink called 'car' before. Are you sure it's a type of food or drink?". This question is again constructed from a textual template, and names of the involved classes ($FooOrDrink$, Car). When user answers with 'no', the system abandons the answer and continues with the KA.

4.7.8 Step 8. What did you order? The second time

Since nothing changed in the KB that would cause the retraction of the

$$ccWantsToAsk(User1, userOrdered(User1, Visit1, x))$$

question that appeared in the step 6, and nothing new appeared in meantime, the system repeats that question. But, as visible in the Table 4.1, step 12, the user is more sincere this time and answers with 'Pizza Deluxe'. See the next step (subsection 4.7.9 for how the system reacts.

4.7.9 Step 9. I've never heard of 'Pizza Deluxe' before.

Now the first thing the engine does to verify this answer is exactly the same as described in subsection 4.7.7 above (query 4.97 doesn't return any results), but the second query ($name(x, "PizzaDeluxe")$ now also doesn't return the results, so 'Pizza Deluxe' is not known to be an inconsistent answer (it can still be), but more likely it's a new type of $FoodOrDrink$. For this reason the engine responds with "I've never heard of 'Pizza Deluxe' before. What kind of food or drink is it?" (step 9 in Table 4.1). Additionally to present the question, it also provides the possible answers (all of the subclasses of the concept $FoodOrDrink$, and additional option "it's not a food or drink."). This allows the user to immediately place the new concept correctly, or reject it and thus behave the same as in the step7 before. In this case user picked the "pizza", which is then asserted into the KB as the following assertions:

$$\begin{array}{c} subclass(PizzaDeluxe, FoodOrDrink) \wedge \\ name(PizzaDeluxe, \text{"Pizza Deluxe"}) \wedge \\ userOrdered(User1, Visit1, PizzaDeluxe) \wedge \\ menuItem(JoesPizza, PizzaDeluxe) \end{array} \tag{4.98}$$

This way the system learned about a completely new concept and placed it properly into the existing KB hierarchy.

Additionally, after this is inserted, the new knowledge triggers the inference rule (rule 4.40), and produces an additional piece of knowledge, that the "Pizza Deluxe" which the user just ordered is also on the menu in Joe's Restaurant: $menuItem(JoesPizza, PizzaDeluxe)$.

4.7.10 Step 10. User2, Where are you?

Now, to produce this step (step 10, Table 4.2), *CC* does exactly the same things as for step 1, but now for a different user $User2$. To see how it constructs, and answer this question refer to subsection 4.7.1. After this visit of $User2$ gets asserted to the KB, *CC* detects that there is already some knowledge for this place which has not been verified, and additionally to start asking questions it also wakes up the *Crowdsourcing Module*. See next step (subsection 4.7.11).

4.7.11 Step 11. Is it true that Joe's Pizza has Pizza Deluxe on the menu?

As mentioned in the previous step (subsection 4.7.10), *CC* finds unconfirmed knowledge on the same topic as is the topic of $User2$, which was provided by other users. One assertion like this is $menuItem(JoesPizza, PizzaDeluxe)$, which it checks with the new user ($User2$), by simply stating it in English and adding a prefix "Is it true that" to the sentence. In this case user checks the menu and answers with "yes", which results in the additional assertion in the KB: $userAgrees(User2, menuItem(JoesPizza, Pizzadeluxe))$, which can be then used by the *Voting Component* to confirm the truth, and also as an additional knowledge about the user (what he agrees with) and used by other inference rules to produce more questions.

Chapter 5

Real World Knowledge Acquisition Implementation

This chapter presents the actual implementation in the system described in the previous chapters (especially in chapter 4). While up to now we've been describing the idea and a general approach to do it (independent of the implementation), here we represent the actual working system that we implemented and kept it online in the shape of the current version since the end of 2012.

Figure 5.1: Screenshot of Curious Cat Android application.

During this time, 5,185 users installed its client app, out of which 2,401 users registered, and 1,715 users used it at least once (see the results chapter, subsection 6.1.5 for more details on users). The development of *Curious Cat* implementation started on

October 26th, 2010 and halted on June 2014. The implementation follows closely the architecture presented on Figure 4.1 in chapter 4. For the core of the system we took Cyc with its common sense KB and inference engine, which was also an initial inspiration for the development and the approach, where the main work needed to be done on the part of the meta-KB and also common sense KB extensions to support our use-cases. For the NL modules, our implementation relies on the internal Cyc logic to NL conversion modules, and on SCG(Schneider et al. 2015). The procedural component and client were written from scratch. All together our implemented system consists of 50,686 lines of Java source code, and 12,616 lines of knowledge definitions (8,571) for the additional knowledge we added in CycKB to drive our KA process.

Besides the implementation described here, this system was also implemented and deployed as a real-time commuting companion(Figueiras et al. 2013), and as an RDF framework for on the field sensor information knowledge acquisition (Bradeško, Moraru, et al. 2012). Commuting companion implementation and this implementation were also described in the books *Intelligent Decision Technology Support in Practice*(Costa et al. 2016) and *Handbook of Human Computation*(Witbrock and Bradesko 2013). This implementation was also mentioned in the *Communications of the ACM* magazine article(Geller 2016).

Implementation consists of Android based mobile client (see screenshot on Figure 5.1), Java Servlet based *Procedural Component* with PostGreSQL database access, two Research Cyc instances (KB, Inference Engine, NL Conversion) to increase the speed and reliability of the system, *Transcript Server* for syncing between Cyc instances, and a web-site for registration and email confirmation. This organization is also depicted on Figure 5.2 below.

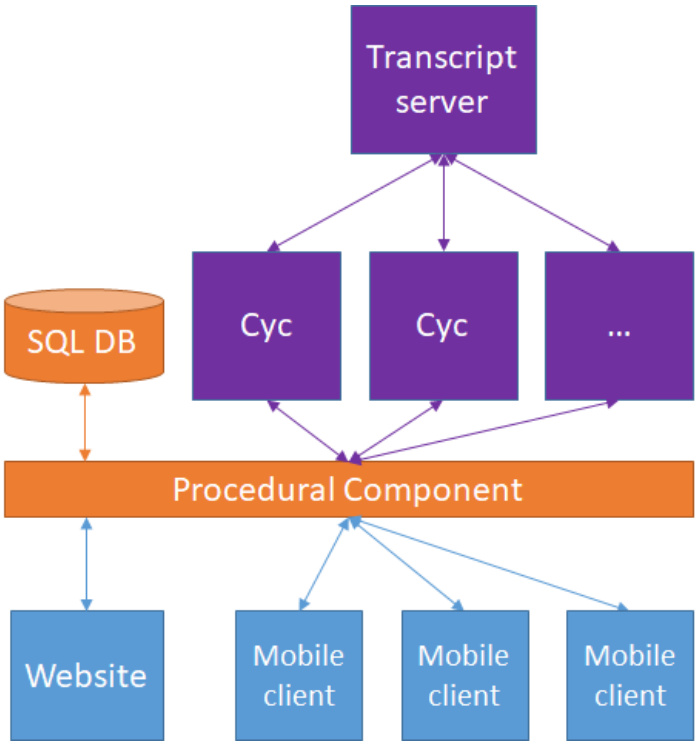

Figure 5.2: Organization of the system modules in our implementation.

5.1 Cyc

As already mentioned in related work (subsection 3.1.1), *Cyc* is an AI project started in 1984 by Douglas Lenat(Douglas B. Lenat et al. 1985) with the premise, that in order to make computers think like humans do, we need to first make sure that they have same common sense knowledge as humans do. Since then, *Cyc* team (now part of *Cycorp Inc.*), is codifying and entering the knowledge by hand and also by other means of knowledge acquisition and mining (see rows stating "Cyc" as parent in Table 3.1, chapter 3 - Related Work). The collected and codified knowledge is represented in machine-usable form using *CycL (Cyc Language)*, and is structured and grouped together as *CycKB*. In parallel with the KB, Cyc also contains an inference engine that works on the scale of the KB, and a set of tools, APIs and other interfaces for interacting with the system, ranging from web interface, natural language, web APIs and also Java interfaces, which we used in our implementation and communication with the procedural part of our system (section 5.4).

5.1.1 Cyc KB

At this moment (end of 2017), *Research CycKB* consists of more than 630,000 concepts, 7,000,000 assertions (rules are also assertions), made with using more than 38,000 predicates (relations), covering a broad domain of human common sense knowledge. The KB is divided into thousands of micro-theories (not counting the microtheories we added for *CC* implementation - section 4.6), where each micro-theory is a set of assertions that share common assumptions. Microtheories (Mts) can hierarchically stack on top of each other and allow *Cyc* to independently maintain assertions that could contradict each-other. This also enhances the performance of the *Cyc* inference engine, since it can be limited on one particular, or a group of Mts to limit the number of facts it needs to use while inferencing.

CycKB is represented in *CycL*, with most basic syntax described as:

- Logical constants are represented by "#$" sign, followed by the name of the term name ($\#\$JoesPizza$).
- Formulas are enclosed in parenthesis. If more than one constants or terms are part of the assertion, the predicate is always first, followed by its arguments, for example: ($\#\$menuItem \quad \#\$JoesPizza \quad \#\$Pizza$)
- Variables are represented by the question mark ("?") sign, followed by the name in capital ($?PERSON$). For example, query asking the inference engine, what is on the menu in Joe's Pizza is represented as: ($\#\$menuItem \quad \#\$JoesPizza \quad ?ITEMS$), where the name of the variable "ITEMS", can be arbitrary.
- Rules in *CycKB* are represented by assertions using predicate $\#\$implies$, where the first argument is then it's antecedent and the second argument a consequent (see CycL rule 5.1 below, which represents the same upper ontology rule as defined in assertion 4.10).

Similar as with our example KB explaining CC approach, *CycKB* is built on top of upper ontology which gives it a formal grounding to support the rest of the KB. Our KA approach (*Curious Cat* system) was implemented using *Cyc*, and also inspired by its toolset and unsolved problems. CC initially started as a solution for Cyc to speed up the NL based KA, so for our explanations it was natural to pick and define an *Upper Ontology* that can be easily mapped to *CycKB*. While we cannot directly map all of the definitions we used to explain the approach, we can map most of the Upper Ontology, which is given in the table below.

Table 5.1: Mapping between upper ontologies of Cyc and our example KB constructed to explain the approach.

CC Upper Concept	**CycKB Upper Concept**
is	*#$isa*
subclass	*#$genls*
arity	*#$arity*
argIs	*#$argIsa*
argClass	*#$argGenl*
Class	*#$Collection*
Predicate	*#$Predicate*
Something	*#$Thing*
Number	*#$NonNegativeInteger*

Similarly as with constants, some rules can be mapped into Cyc versions by replacing implication form ($x \implies y$), with the CycL form ((*#$implies* ($x$) ($y$)). For example, rule 4.10 is in *CycL* written as follows:

$$\begin{array}{l}(\#\$implies \\ \quad (\#\$and \\ \quad\quad (\#\$genls \quad ?X \quad ?Y) \\ \quad\quad (\#\$genls \quad ?Y \quad ?Z)) \\ \quad (\#\$genls \quad ?X \quad ?Z))\end{array} \tag{5.1}$$

5.1.1.1 Curious Cat KB

Even though *CycKB* is one of the biggest and most complete common sense KBs that currently exist, it still does not cover everything and has missing knowledge. While it covers at least a bit of almost any topic within the general knowledge, it often has missing parts that need to be extended in order to use the knowledge in real world. While this is exactly what *Curious Cat* is helping to solve, we also added some of the knowledge by hand, to make initial questions more interesting and to make it cover more topics. For the whole CC implementation, we added an additional 8,571 assertions on top of the existing 7mio assertions of *CycKB* prior knowledge.

- New assertions are mostly KA meta-knowledge supporting the acquisition mechanisms and logic.

- Some of the new assertions are also missing general knowledge improvements on top of CycKB.

For the initial *Cyc* knowledge entry process, we used ".ke" files, which support bulk import bigger amount of the assertions (see Figure 5.3). After the system was already stable and running all the time, this was replaced by the Transcript Server syncing as described below in section 5.2.

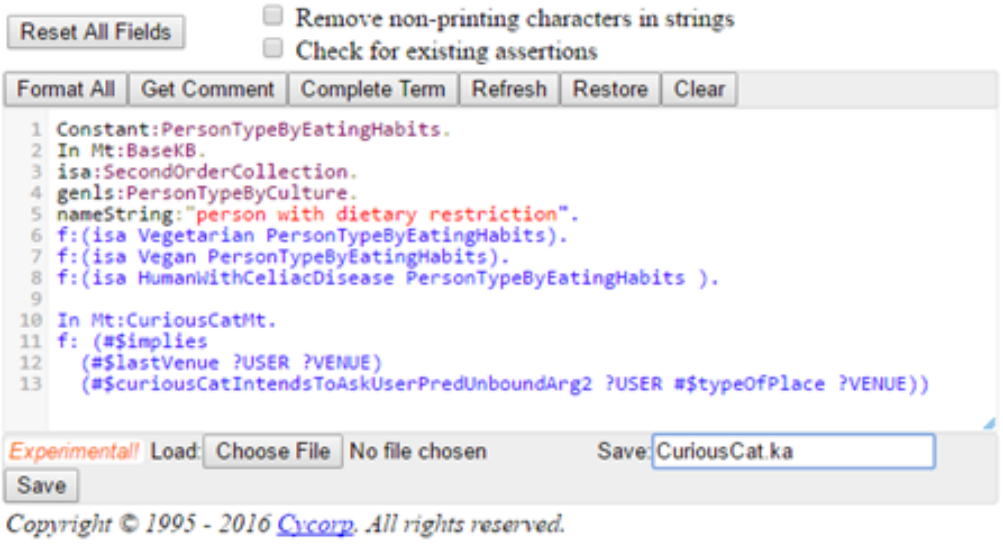

Figure 5.3: Example screen of bulk importing assertions (knowledge) into Cyc. This example is defining a concept $PersonTypeByEatingHabits$ and a rule generating questions about $typeOfPlace$ for last known user venue visit.

While mapping from our explanations to the actual *Cyc* upper ontology is pretty straight forward since it is small, it would be impossible to describe all details of more than 8,000 assertions that define the implementation of CC KA approach. For example, the simple $ccWantsToAsk$ predicate (see definition 4.17 chapter 4, alternative in the real-world implementation is 8 predicates (see definition 5.2 below), allowing us to define much more complicated statements and enriching them with possible suggestions as visible on Figure 5.4. Similarly, the KA rules are more complicated, using much bigger vocabulary than we were able to explain. Example of one such rule is presented on Figure 5.5.

On top of 7 million of preexisting *Cyc* assertions and 8,571 of manually added CC assertions, our system collected from users additional 524,276 assertions (as of September 2017).

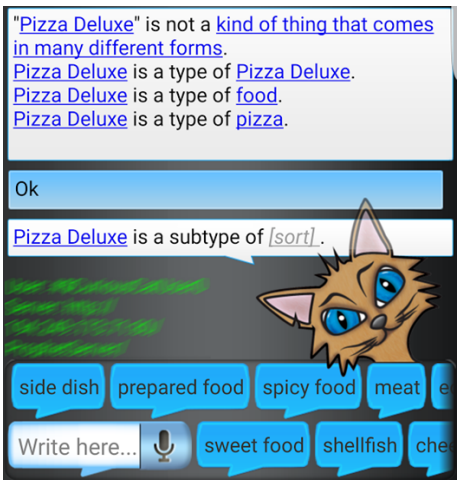

Figure 5.4: Example of human to computer interaction options generated with $\#\$curiousCatIntendsToAskUserPredUnboundArg2WithChoices$.

```
(implies
   (and
      (lastVenue ?USER ?VENUE)
      (isa ?VENUE ?RESTAURANTSPECIALIZINGINCUISINEFN)
      (suggestionsForCuriousCatQuestionType
         (RestaurantQuestion-CuisineFocusedMenuItemFn ?CUISINE) ?SUGGESTIONLIST)
      (termOfUnit ?RESTAURANTSPECIALIZINGINCUISINEFN
         (RestaurantSpecializingInCuisineFn ?CUISINE)))
   (curiousCatIntendsToAskUserPredUnboundArg2WithChoices ?USER restaurantHasMenuItem ?VENUE ?SUGGESTIONLIST))
```

Figure 5.5: Example of KA rule from CC implementation (screenshot from Cyc).

$$
\begin{aligned}
S_{CCIntents} = \{ & \\
& \#\$curiousCatIntendsToAskUserPredBound, \\
& \#\$curiousCatIntendsToAskUserPredUnboundArg2, \\
& \#\$curiousCatIntendsToAskUserPredUnboundArg2WithChoices, \\
& \#\$curiousCatIntendsToAskUserPredUnboundArg3WithChoices, \\
& \#\$curiousCatIntendsToAskUserYesNoQuestionAboutVenue, \\
& \#\$curiousCatIntendsToAskUserYesNoQuestionAboutVenueWithArgument, \\
& \#\$curiousCatIntendsToAskWithChoices, \\
& \#\$curiousCatWantsToAskUser \\
\} &
\end{aligned}
\tag{5.2}
$$

5.1.2 Cyc Inference Engine

Cyc AI system also includes the inference engine which can work over the CycKB. Because the KB is so big (more than 7 million assertions), approaches taken by other inference engines (like RETE algorithm) does not scale well at this size, while the included engine is with some tweaking still able to give results inside tolerable time frame. One of the techniques (mentioned before) allowing this, is organization of the KB into micro-theories which allow the inference engine to reduce the size of the world it operates with at given contexts.

Cyc Inference Engine is able to perform modus ponens and modus tollens inference, universal and existential quantification, and also mathematical calculations[1]. The inference engine is constructed together with a set of *Inference Removal Modules* (each module is able to remove or solve a specific problem from the inference problem). These modules can range from very general, to very specific ones, registered to only solve problems related to one particular logical predicate. For example, predicate $\#\$disjointWith$ can have a module that can tell whether the assertions using this predicate are true or not (decides on disjointness). There are many modules like this, implementing symmetry, transitivity and reflexivity, etc.

The engine is mostly controlled with rules (as for example the rule on Figure 5.5). The inference rules can be separated into the forward and backward rules, which define whether they are to be used by forward chaining or backward chaining while doing the inference. Forward rules are triggered when something is asserted into the KB and then they produce the results (assertions as defined in the consequent), while backward rules are triggered

[1] http://www.cyc.com/subl-information/introduction-cyc-inferencing/overview-cyc-inferencing/

when a logical query is being asked over the KB, and only support the logic behind the answering, without actually physically populating the KB. The example KA rule from the Figure 5.5 is forward rule.

Forward rules are causing the system to be slower when something is being added, and also cause its memory consumption to increase (newly inferred assertions), but on the other hand speed up the querying. Backward rules are the opposite, not taking any burden on the system when the assertion is being added, but trigger the inference which could be slow, when someone issues a query.

5.1.3 Cyc NL

In the subsection 4.4.1 and subsection 4.4.2 we gave a simplest possible example of logic to NL and the opposite conversion, to explain and showcase the approach taken by the *Curious Cat*. While this served well for the explanation, the implementation in real-world is inherently more complicated. For the implementation we used Cyc logic to NL system which is part of *Research Cyc*(Coppock et al. 2010; Baxter et al. 2005), and is able to convert a decent number of *CycL* assertions into their natural language equivalents. In our example, we only used two (2) predicates to be able to do that (*name*, *nlPattern*), but the implemented system uses more than 90 predicates and functions that are used to make the paraphrase system to work. On the Figure 5.6, we can see the screenshot of NL generation assertion using 4 paraphrase functions, unbound variables, a string and a concept of the $Have - TheWord$. Then additionally, the concept representing the word "have", has additional 79 language generation assertions defining how to convert this word in various discourse contexts.

Figure 5.6: Actual NL generation assertion example for the corresponding predicate for *menuItem* from our implementation (Cyc).

As part of implementing *CC*, we had to extend this system and add multiple NL generating assertions to it, the implementation is from *Cyc AI System*, and is out of scope of this thesis. The details of this approach which can generate both referring expressions and non-referring expressions, including both referential and bound variable anaphora, are described in the paper written by Coppock et al. (2010).

To be able to convert from NL back to logic, our implementation also relies on the functionality which is part of *Cyc* and is described in more detail in the Semantic Construction Grammar paper(Schneider et al. 2015). The approach is similar as taken by chat-bot engines(Wilcox 2011), which uses textual patterns with special wild-cards, to map the actual text onto a predefined pattern, which is then used to find a response. The difference is, that after the initial pattern matches are found, it uses *CycL* mappings and then Inference Engine to check for the formal validity of the resulting statements and consequently the

parsing process. The other improvement is, that the the wild-cards in the patterns can be typed (using Cyc constants). This means that that in such patterns, only terms of specific type can be fitted into a slot, which improves the conversion accuracy. An example pattern (taken from the paper) can be written like: "place[d|{}] $ContainerArtifact#0 over high heat". During the years of CC experiment (as described in the chapter 6), the logic to NL was used all the time (it is a crucial component), while NL to logic was implemented but not always used, because it is more complex part of the software, harder to maintain, and was also not of crucial importance for KA experiments.

5.2 Transcript Server

The *Transcript Server* is an important technical part of the presented CC implementation, improving scalability, reducing downtime and also reducing other maintenance costs. As visible in the implementation design (Figure 5.2) sketch, it is a crowd based software module to which all Cyc instances connect to. Software wise it is a simple database server, which can receive and store for later, raw *CycL* assertions (without any inference capabilities), and replicate them between connected *Cyc* instances. Each of CC instances of *Cyc* system is set in a way, that after it wakes up, first syncs with the *Transcript Server* and thus gets the knowledge added in the past. Besides the syncing, it also forwards every new assertion to the *Transcript Server* (if it was not received from there).

This simple mechanism allows us to have multiple *Cyc* instances with the same knowledge, which then allows simple load-balancing at query-time, backup instances in cases when the main instance crashes. For the cases when we had to manually shut down Cyc for upgrades or server maintenance, this simple automatic syncing mechanism (even if being slow due to each assertion triggering the inference engine again), saved us a lot of manual importing of the old knowledge. As of September 2017, full sync of CC collected knowledge (532,847 assertions) takes approximately 2 days to finish.

5.3 Mobile Client

The client part of the implemented KA approach is a simple Android application (see screenshot on Figure 5.1 at the beginning of the chapter). The main screen consists of (listed top down as sections appear on the phone screen):

- Upper status bar, containing the points on the left, location info and button in the middle, ant a "cat" menu on the right.
- Middle conversational section, where CC and user responses appear as text in the conversational bubbles, similar to those that usually represent SMS messages. Each bubble can contain HTML formatted text, hyper-links and pictures.
- Conversational section is followed by the "response section", where user is presented with the set of possible responses, or a free text box. On the right of this GUI section, there is a cat's face, which acts as a button with same functionalities as the "cat" menu on the upper right.
- On the bottom, there is a swipe-able section, which contains contextual bubbles representing what other users who are nearby or are discussing same topics are answering.

The logic behind the client is pretty straight forward. It is connected to the *Procedural Component* of CC system, which can send it a discourse object containing a text which

needs to be presented to the user, and some additional info such as ID of the statement, it's *CycL* representation and also suggestions for the user, and the points that user currently collected. When the user answers, this is sent back to the server, together with the ID of the discourse. In the cases, when the server wants to pro-actively ask something, and the phone is in a sleep mode (meaning that the pro-activity didn't occur due to location change), it can generate a FCM message with the same object as it would send directly, and then this is presented to the user when he/she looks at the phone.

Additionally to this answer/reply functionality, represented as blue arrows on Figure 4.1, the client keeps a back-channel with the server's *Procedural Component*, and is periodically sending a location, local time and language. In order to not consume a lot of the battery, the client is mostly sleeping, and only wakes up every 3min and tries to get a GPS signal for 20 seconds. If the signal is too low and cannot acquire accurate enough location in 20 seconds, it stops trying and goes back to sleep mode. The back-channel is also opened each time user opens the client application manually.

5.4 Procedural Component

This component serves as the main interface between the *CC* system and the external world, and as a glue, keeping all the parts of the system in sync and together as a coherent KA platform. This component consists of a database, two Java servlets, and a separate Node.js server handling the geo-spatial context mining functions. Each of the servlets is handling one type of interaction with the client applications:

- *Main Interaction Servlet* is taking care of the user sessions, and main interaction loop between the user and the system. This servlet is handling most of the work when user is receiving and answering the questions. It also asserts and detects new topics, when user clicks hyper-linked concepts that are presented as part of the conversational bubbles.

- *Location Servlet* is accepting GPS records from the phone, calling the geo-spatial clustering service and then asserting current location and time assertions into *Cyc*, as described in section 4.3. Additionally, this servlet is providing the contextual clues (bubbles at the bottom of Android app. as visible on Figure 5.1) to the clients.

5.4.1 Main Interaction Servlet

When this component gets the request from the client, it first checks whether it is linked to one of the statement IDs that were sent to the client. This identifies, whether this is an answer to some previous questions that CC asked, or is a completely new discourse. This identification can be done on the client already, since the GUI is constructed in a way, that it is obvious to the user, he is answering a question. If he wants to do something completely independent, the guided mechanism "forces" the user to click on "something else", where he can then start a completely new discourse. After this is identified, the system asserts into *Cyc* and logs in the database the time of last user response, and text of the response. After this, depending on whether it was identified as an answer to previous question, or a completely new discourse (statement, question), it goes into the *Consistency Check and KB Placement* mode (section 4.5), or *NL to Logic* mode (subsection 4.4.2), as described in the appropriate sections in the chapter 4.

After this step, when the system already remembered the user answer, or understood user query, this servlet asks *Cyc* for new questions and comments that *Curious Cat/Cyc* might have. This is done with logical queries like this (let's say that the current user is

$User1$): ($curiousCatWantsToAskUser$ $User1$ $?QUESTION$ $?SUGGESTIONS$). This will then cause he inference engine to search through all the assertions and trigger all the backward rules in the micro-theory KB of $User1$, and come up with the answers, filling the variables $?QUESTION$ and $?SUGGESTIONS$. The list of questions (given in logic, *CycL*) is then analyzed by the servlet, which picks the first one that was not there before (on previous interaction), and mentions the concept that is set as a current topic. Topics are organized into a stack, where the concepts user is mentioning when answering questions about the current topic, are constantly being added on top of the stack (Table 5.2). In addition to these "user caused" topics, stack contains a three predefined topics at the bottom. First one (last to be used - at the bottom), is a random topic that gets chosen by the system. Second "topic" is actually a question, whether there is some particular topic that would interest the user. The third one (first of the system's topics, to be used when it runs out of the questions), is the concept of the user itself. If it happens that *CC* runs out of other topics, it starts discussing about the user.

Table 5.2: Stack of conversational topics.

$\#\$CurrentTopic$
$\vdots$
$\#\$AnswerConcept2$
$\#\$AnswerConcept1$
$\#\$PreviousTopic$
$\#\$PreviousConcept1$
$\#\$UserConcept$
$\#\$AskUserForTopic$
$\#\$Random$

For example, on Table 5.2, we can see that the current topic is taken out of the stack (separated by the double line), and the other concepts that were in the answers of the questions, were added on top ($\#\$AnswerConcept2$). We can also see, that the topics are added back on the stack, if their questions are not depleted yet (as happened for the $\#\$PreviousTopic$). For example, let's say that the $\#\$PreviousTopic$ was *Joe's pizza* restaurant, which was on the stack right above the $\#\$UserConcept$ topic. It was taken out of stack, and then when user answers the first question (let's say "they have pizzas on the menu" as in our other examples), concept $\#\$PreviousConcept1$ represent $\#\$Pizza$, and is added on the stack. Then user decides that she wants to discuss about some other topic ($\#\$CurrentTopic$), and this causes the $\#\$PreviousTopic$ to be put back on the stack, since not all questions were depleted. But now, it sits on top of the "Pizza" topic ($\#\$PreviousConcept1$). The $\$\#AnswerConcept1$ and $\#\$AnswerConcept2$ were then added on top of the stack, as a consequence of answering questions inside the $\#\$CurrentTopic$ topic. An additional fail-back mechanism (not presented on Table 5.2), for when the topic is depleted of the KA questions, is to call internal *Cyc CURE* API (Witbrock 2010), to get a list of general, or internal *Cyc* questions (not using any context) that could be asked for particular concept.

5.4.2 Location Servlet

When the client sends the GPS location and user local time (either when periodically waking up, or when user turns on the application), the coordinate gets into a clustering algorithm which detects whether this is part of user moving between the locations, or is

part of the visit of particular location. In the case, when user is moving, this is asserted into *Cyc* as the user activity, but no particular action is taken. This simply influences the *CC* responses in cases, when user interacts with the system while he is on the way. On the other side, when the algorithm detects that user is at particular location, this goes to the second algorithm, which checks online APIs(Factual, Foursquare) for the type and name of the place. This information, together with the time of arrival and current stay, is asserted into *Cyc*, together with the new topic, representing the concept of the new location. At this point the system "knows" the user is staying at particular place and might have some free time, even if the client application is sleeping, it sends a FCM message with a new contextual questions, which appears as a notification on the user's phone.

The first part of the clustering algorithm is an improved implementation of already mentioned (subsection 3.5.1 in chapter 3 and described in subsection 4.3.1 in chapter 4) SPD (Staypoint Detection) algorithm(Kang et al. 2005). Our improvements and extension to the algorithm(Kazic et al. 2017) improved the accuracy and robustness by a large factor[2], and can be quickly summarized:

- Removing some unnecessary steps performed in the original algorithm (step 13 from the paper written by Kang et al. (2005).
- Extending the algorithm to also reports locations that are not detected as a point of stay and thus allow it to be an online algorithm, working in real-time, quickly responding to possible location changes and reporting whether the user is currently traveling or at a fixed location. This is achieved by keeping the index of locations, until completely clustered. The paths are then locations from this index, not included in any of the clusters. Additionally, the current cluster (marked as CL on the algorithm 3.1), is returned each time as well, marked as unfinished.
- Adding an optional second iteration of the algorithm which fine tunes promising detected stay-points from the first iteration. This cleans the potential errors, when one or a few raw GPS locations jumped outside the perimeter but then the sensor stabilizes. This adds another loop through the detected clusters at the end of each iteration. It creates one visit (stay-point) on the occasions when the path between two same locations lasts less than a given threshold, which is the most common mistake appearing in the results of the original algorithm.
- Adding an additional analytics steps, including enrichment of the detected stay-points and predicting user's next destinations.

This algorithm through the years while *CC* is online evolved from a few "if" statements running locally on the phone, to a separate service and mobile library serving also other projects, due to it's general usefulness for geo-spatial analytics. The data that the clients (not just *Curious Cat* are sending and the service is analyzing, can be visualized and browsed through a separate web-interface as visible on Figure 5.7.

The second part of this algorithm was inspired by the paper *Applying Commonsense Reasoning to Place Identification*(Mamei 2010), which probabilistically ranks the possible places acquired from online APIs based on the SPD results and current time of the day. This then additionally enriches (with the name and type of the place) raw geo-coordinates of the cluster and time as given by the SPD algorithm alone.

Additionally to handling user current location and time, *Location Servlet* also uses this context, together with the current topic (Table 5.2), to query for latest non-private answers

[2]An additional paper with the full explanation and evaluation of extended SPD algorithm that can be used in any location-based application is in preparation

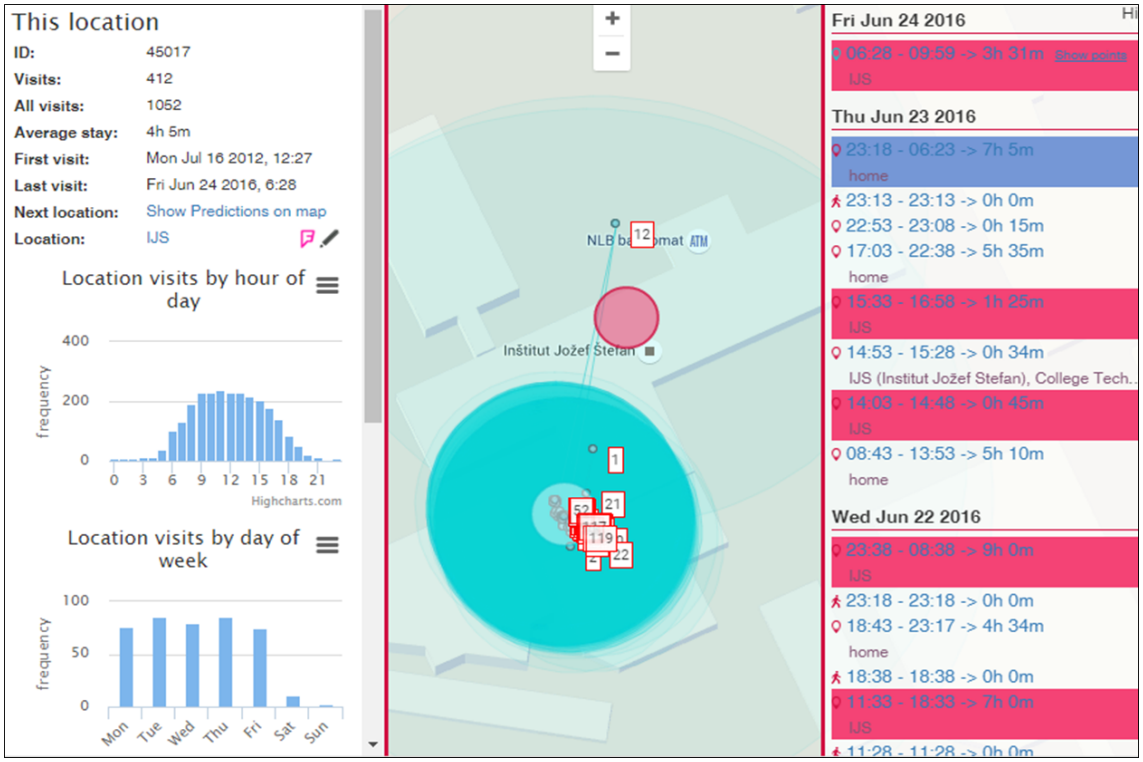

Figure 5.7: Visualization of clustering and enrichment results the SPD algorithm, to provide *probableUserLocation* context.

from other users that are nearby, or talking about the same concepts as our example user. This is represented by contextual clues at the bottom of the client application (Figure 5.1).

5.5 Assisting and Using Gained Knowledge

One of the main ideas behind proposed KA approach, is to be able to incorporate it into some other activity, and do the KA unobtrusively and cheaply as a side effect of the main purpose of the software. This was also our initial aim when we started the *Cyc* based implementation of proposed KA system. The end goal of the KA is thus not the KA itself, but to be able to collect good enough knowledge to be able to use by the inference to solve some higher-level tasks previously not possible. To prove this concept, we added a few inference rules which try to combine all the gained knowledge to produce relevant suggestions when user requests them, or when the system thinks the user is hungry (no eating reported for a while). Additionally, the system is able to infer, based on the events around the user and his likes (answered questions about what user likes).

On the left screenshot of Figure 5.8, we see that *Curious Cat* smartly told the user that there is a (jazz) music event nearby, after it learned that the user likes jazz music. The event was actually entered by another user and the system picked it up from the KB for the purpose of helping new user. On the right screenshot, there is a similar ad-hoc comment, but this time the trigger was cheaper coffee in the bar next-door. The suggestion is immediately followed by the comparison question, to be able to better rank the coffees next time.

A more complex inference suggestion when user asked for something fast to eat is pre-

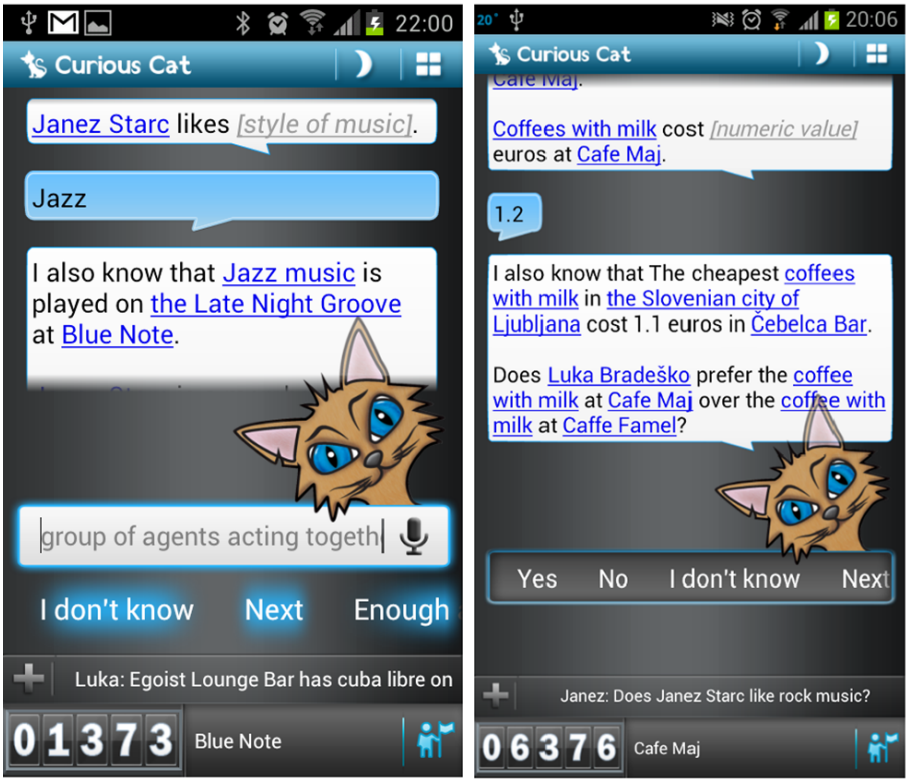

Figure 5.8: Example suggestions for a live music (after it learns that the user likes jazz), and information that there is a cheaper coffee nearby, followed by the comparison question to be able to suggest better in the future

sented on the Figure 5.9. There the inference engine combined multiple pieces of knowledge:

1. Hot horse serves fast food cuisines, therefore it must be a fast food restaurant
2. User likes fried food
3. French fries are a type of fried food
4. Hot Horse has French fries on the menu
5. Since Hot Horse is a fast food restaurant which is near the user, and has something on the menu that the user likes, it is among the good suggestions.

This inference process was then structured into the set of logical statements (special inference rule for suggestions), sent through the NL conversion and presented to the user as the suggestion text. While this suggestion is a simple basic example, still shows the power of combining separate piece of logic by inference and produce some useful results. The step from liking the fried food to French fries on the menu would be quite hard to tackle with standard approaches, especially if we consider it just an example of a general mechanism.

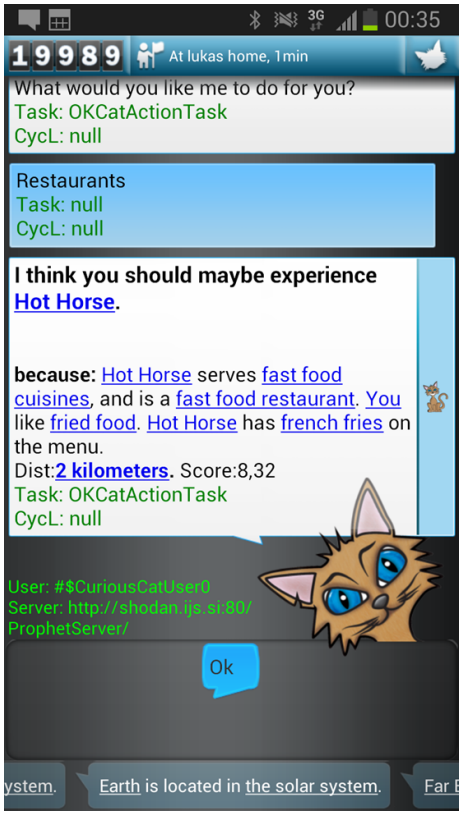

Figure 5.9: Suggestion as a complex inference result combining knowledge about cuisines, user likes and place specifics.

Chapter 6

Evaluation

Validation of the proposed approach was performed using our *Curious Cat* implementation of the proposed KA approach, running alive over the course of 4 years engaging a total of 728 registered users (users which are part of the experiment. Overall the implementation have some additional users). During that time, the users checked-into 5,551 locations and responded to 57,978 questions, out of which 8,611 were voting questions (7,560 positive and 1,051 negative votes), 18,907 questions were answered with "I don't know", and 30,460 real answers was inserted into the KB as new knowledge, including 31,140 concepts (3,171 connected to instances of *User* concept, 22,563 check-ins and other places, and 5,406 other concepts). These triggered additional 386,980 assertions to be added through forward-chained inference. These can be separated into facts (374,300 assertions) and additional derived question generating rules and questions (12,680). Altogether we gathered **444,958 (374,300 + 30,460)** pieces of completely new knowledge.

For easier understanding of the results, we show the structure of the collected knowledge graphically in Figure 6.1, and give real examples of the collected and inferred knowledge in Table 6.1.

While the resulting number and quality (Table 6.7, Table 6.8) of assertions shows that the presented approach can be successfully used for high quality KA, we explored in more detail the contributions of specific characteristics of our system and compare them to the baselines, to evaluate the ideas and claims of the approach:

1. Using context to pro-actively drive the KA increases engagement and the chance of getting an answer (section 6.1)

2. Pre-existing knowledge and automated inference can be used to filter-out inconsistent answers and thus increase the quality of the acquired knowledge

3. Using newly acquired knowledge to further drive the KA process increases the amount of acquired answers and reach of the system

4. Crowdsourcing additionally improves the results by filtering wrong but logically correct answers

6.1 Context and Pro-activity

Curious Cat employs two contextual knowledge provision mechanisms. One is location based knowledge, such as, the type of current location, the time of visit and the duration of stay. The second is internal knowledge that the system already knows about the user, such us, languages that the user speaks, food she likes, demographics data, interests, etc.

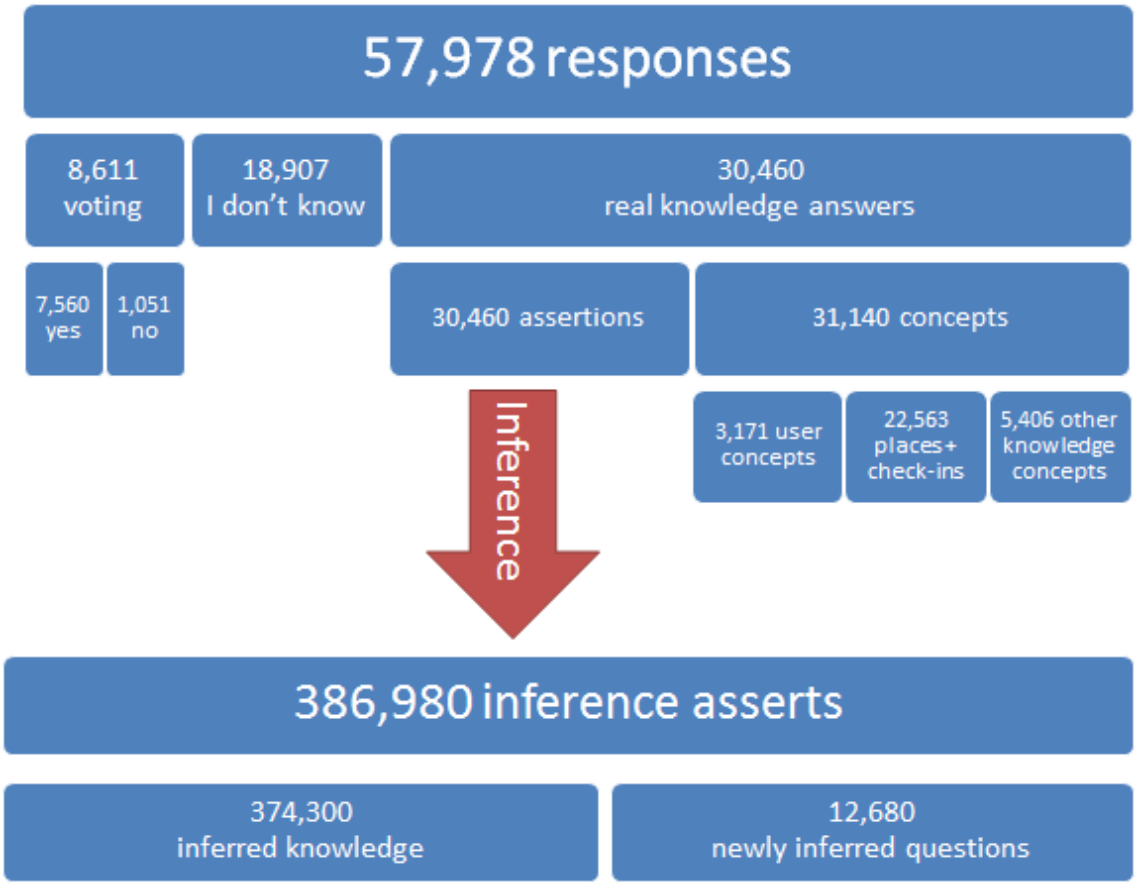

Figure 6.1: Graphical representation of collected answers (knowledge) and distributions.

In evaluation, we focused on exploring benefits of using the location based context, since it is practically impossible to disable internal knowledge without bigger changes in the system. Inferring without internal knowledge e.g., total removal of the personal knowledge would mean removal of almost all questions regarding people, animals, human interests, etc., since these are general for human beings and to some extent to animals as well.

For the purpose of this experiment, we deliberately removed GPS location and connected mechanisms for derived knowledge for a duration of three months. We normalized the measures (assertions per user per day) for all experiments to level out the differences in the durations of experiments against a longer duration of fully operative system. The results are presented in Table 6.2. The columns in the table are organized as follows:

- first column contains measure name,
- second column full data-set when using context (marked as C),
- third is 100-day data-set when not using context (NC100),
- and last is the normalized column with context (C100) which makes it possible to compare the context versus no context knowledge collection behavior (C100 vs NC100).

Since number of days and users is strongly correlated with the number of answers the system gets, and we only have the "No Context" data for 100 days (41 new users during

Table 6.1: Examples of answered/asserted and inferred knowledge taken from *Curious Cat* KB.

Type of the collected knowledge	**Curious Cat Question**	**User Answer**
User Concepts	[automatic assert from registration]	$is(User1, Person)$, $name(User1, "Luka")$
Places + Check-ins	[automatic assert from check-in and Foursquare/ Factual locations]	$is(Place1, Restaurant)$, $name(Place1,$ "Pig'n Whistle")
Other Concepts	What did you order?	Duck meat
Real Knowledge 1	Who is CEO of BMW?	Harald Kruger
Real Knowledge 2	What is the ticker symbol of BMW?	ABC
voting(yes)	Is it true that Herald Krueger is the CEO of BMW company?	yes
voting(no)	Is it true that ABC is the ticker symbol for BMW?	no
I don't know	___ and BMW are corporate competitors.	I don't know
Inferred knowledge 1	$is(HeraldKrueger, Person)$	
Inferred knowledge 2	$anatomicalParts(HeraldKrueger, Hand)$	
Newly Inferred question	"What is Herald Krueger's age?"	

that time and 45 active users), we normalized our "Context data" in such a way, that it matches the duration, new users and active users of *NC100*. For this, we had to scan all the data-set day-by day with a range of 100-day window, where we looked at the number of new and active users in each of the 1264 sub-windows (number of all different 100 day windows one can fit into 1264 days). In the whole data-set there was 18 such 100-day windows, which can be directly compared to the *NC100*. While the number of new users is fixed to 41, number of new assertions and active users still varies (min. 1,429, max. 9,279 new assertions and min. 47, max 61 active users). For this reason, we calculated the mean values of all 18 windows which had the same number of new users as *NC100*. Matching new users (instead of active users) was selected because new users are the ones that contribute most of the assertions, since their interest winds down gradually. This can be seen on Figure 6.2. This is also the primary reason why (unintuitively) number of new assertions/day/active user (row 7 in utoreftab:ccresultscompare) is higher without context (*NC100*) than for non-normalized data from experiment using context (C).

The results of not using context show the decrease of raw new knowledge assertions from 56,568 to 709, which of course can be attributed to the fact that duration of experiment C was much longer, or that it had more users than *NC100*. As described above, this is properly normalized in the column *C100*, where the raw number of assertions raises to 2,244 compared to only 709 when not using context (**context brings 217% increase**).

While the majority of the increase is due to pro-active questions from *Curious Cat*, which are linked to GPS clustering, some of the increase is also due to better targeting of the questions due to contextual data, since without the context only users are selecting the topics and the system doesnt have enough knowledge to successfully pick the most relevant

Table 6.2: Results of KA without context versus results while using location based context

Measure	**With Location Context (C)**	**Without Location Context (NC100)**	**With Location Context 100 days (C100)**
1. Experiment duration	1,364 days	100 days	100 days
2. Number of new assertions	56,586	709	2,244 (+216.5%)
3. Number of new assertions where user didn't know the answer	18,380 (32%)	267 (38%)	667 (28.3% = -9.7%)
4. Number of new assertions where user knew the answer	38,206 (68%)	442 (62%)	1,577 (71.7% = +9.7%)
5. Number of new assertions/day (all users)	42.1	7.1	22.4 (+215.5%)
6. Number of new assertions/active user	90.3	15.7	44.2 (+181.5%)
7. Number of new assertions/day/active user	0.07	0.16	**0.44 (+175%)**
8. Number of new concepts/day	3.7	0.4	2.4 (+85.2%)
9. Number of new concepts (excluding users and locations)	4,925	36	243 (+85.2%)
10. Number of active users	625	45	49
11. Number of new users	625	41	41

next question. This is reflected in a slight increase in the proportion of the questions for which the user knows the answer (+9.7% - row 4 in Table 6.2). This is independently of pro-active component which is mostly reflected in the raw number of assertions, while the knowing/not knowing ratio improvement is due to better targeting whatever question there was.

6.1.1 Consistency Checking

Before the system accepts the answer from the user (as described in section 4.5), it converts it into logic and checks whether it is consistent with its current knowledge. If there is an inconsistency, it will reject the answer and thus prevent the wrong or contradicting knowledge to corrupt the KB and consequently the future KA process. The user then either has to fix the answer, or convince the system that the answer is actually only a different name for the consistent knowledge. For example, if the user states that he ate a car in the restaurant, this is either wrong, or car is the name for some unknown food and not a vehicle.

Due to development process and time-line of the system, we only have logs and consequently insights on when/how the system rejected the answers due to inconsistency for 143 out of 1,472 days of the experiment duration. During this time, 148 users provided 23,543 answers and the system rejected 563 of them, so 22,980 of assertions went into the KB. Among the 563 rejections, some of them were repeating (user re-tries, or other users

did the same mistake), so the system actually prevented 384 unique inconsistent answers.

Table 6.3: Results of Consistency Checking

Measure	**Real data from the logs**	**Extrapolation for the full experiment**
Number of new assertions	22,980	57,978 days
Number of rejected assertions due to inconsistency	563	1420
Percentage of rejected asserts	**2.45%**	**2.45%**
Number of unique rejected asserts	384	968
Percentage of unique rejected asserts	**1.67%**	**1.67%**
Number of new users	128	728
Number of active users	148	728
Experiment duration	143 days	1472 days

As presented in Table 6.3, for every 100 valid answers users provided during the experiment, there were 2.45 answers which were rejected by the system due to being completely wrong or inconsistent. Because we only have logs for 143 days we had to extrapolate the number for the overall experiment to get some estimation of how many bad assertions were prevented by the system throughout the whole experiment. The estimated number of rejected assertion system prevented overall is 1,420. Fore easier understanding, in Table 6.4 are examples of the rejected assertions, together with the interaction that lead to the rejection.

Table 6.4: Conversation prior to rejected assertions

CC	**User**
Example 1	
Which social being has Lagos as assets?	hotel
By hotel do you mean hotel (building)?	yes
Rejected - building is not a social building	
Example 2	
Let's continue our previous conversation about going on a date.	ok
All going on a date has ___as a characteristic sub-event	hello
Rejected - hello is a greeting, not an event.	

6.1.2 Additional knowledge as follow-up of newly acquired knowledge

As one of the more important features of our proposed system is its ability to use newly acquired knowledge to ask new things, we explored this functionality, comparing it to the

baseline KA and consequently measuring the knowledge that was acquired as a consequence of prior acquisition. An example of such a follow up question or assertion is given in Table 6.5.

Table 6.5: Example of follow-up question and answers (new knowledge)

1	CC Asks: "What did you order?"
2	User answers: "Meatloaf"
3	CC asks a follow-up question: "One of the meatloaf ingredients is ___".
4	User answers: "egg".

The question number 3 in Table 6.5 is a "follow-up" question since it is only asked when someone enters the previously unknown food concept. Answer under number 4 is a follow-up knowledge, since it was answered and given to the system only because someone created the concept of *Meatloaf*.

During the whole experiment the system collected and accepted **22,838 answers**, which were a follow-up answer. This means that **39.4%** of all the knowledge we obtained was a direct consequence of the systems ability to generate new questions based on the already acquired knowledge.

The follow-up questions and answers are always a consequence of some new concept that enters the system. The 22,838 of answers are consequence of 5,406 new concepts (not counting concepts of users themselves and locations they checked in), where the distribution of new knowledge/concept follows a long-tail distribution - a very small number of concepts triggered a big number of follow-up knowledge (Figure 6.2 - logarithmic scale).

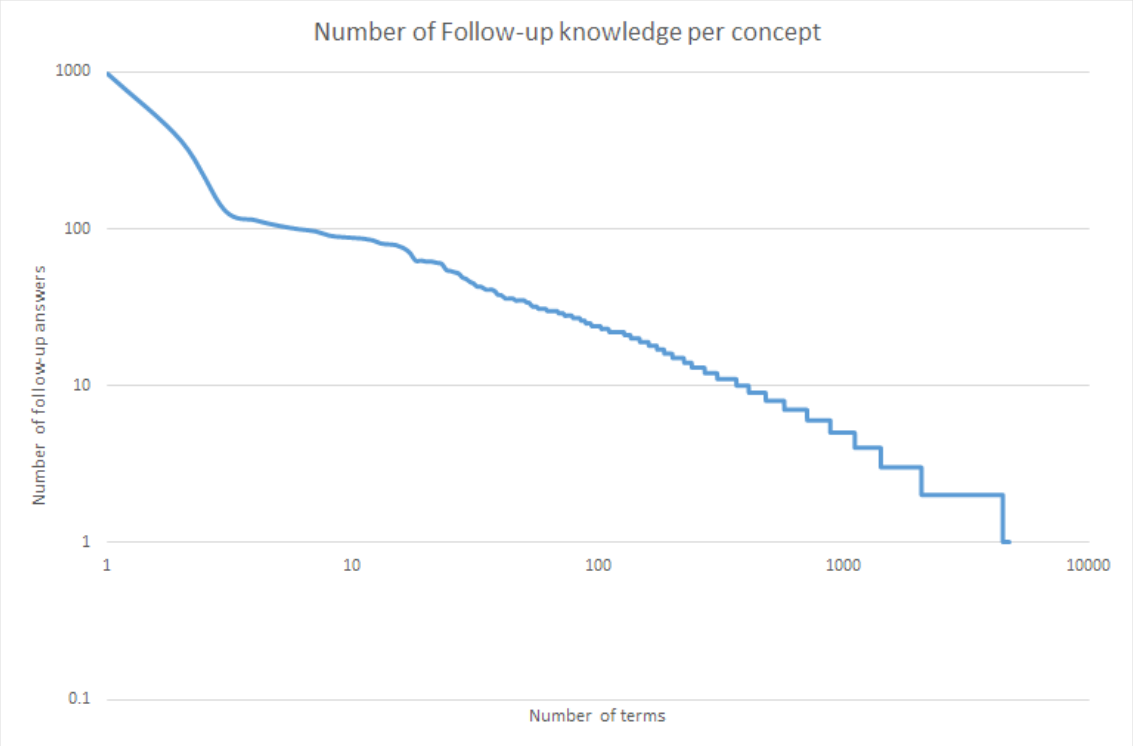

Figure 6.2: Long tail (logarithmic scale) distribution of follow-up answers per new concept

6.1.3 Additional knowledge because of allowing cross-user cooperation

Similarly, as done for follow-up answers in subsection 6.1.2, we can check how much of the new knowledge came into the system due to its ability to share knowledge and questions cross-users. For this, we check the answers, where the initial concept was given to the system from a different user then the follow up answer. From the example in Table 6.5 this would mean that the answer number 2 was given by $User1$ and then CC asks the question 3 and receives answer 4 from $User2$.

While for follow-up results, we have a complete log, due to changing the system, not all records have information about the original user who created the concept. We have his data only for 45 concepts out of 5406, excluding all user concepts and locations they checked-in. For these 45 concepts, we got additional 123 answers from other users.

6.1.4 Voting

The voting part of the system is, besides directly providing the answers, one of the main crowdsourcing components. As described in section 4.6, after the knowledge is acquired, the system can re-check with other users whether the assertion is true or not. This is done by presenting a question in the form: "Is it true that [assertion].", which can be answered with "yes" or "no". For example: "Is it true that Bornholmsk is a national language of Denmark?".

While the consistency check is performed automatically by the system, the answers that are manifestly inconsistent with the other content in the KB are stored only in the user specific part of the KB. Checking for truth of the knowledge is motivated by the fact that claims can be still be untrue even if they are consistent with the KB. From the Yes/No ratio, we can see that there are many more "Yes" votes than "No" votes, which hints that the answers the other users provide are mostly recognized as true by the crowd. This could be taken as a hint towards the precision of the truthfulness of the acquired knowledge. If we take into consideration that more users can vote for the same assertion, the effective precision of the knowledge measured this way can be estimated more accurately. The voting mechanism hides from the public knowledge base 97 of the assertions where the users were unable to agree on truth and 636 assertions which were voted as untrue by majority of the users.

Table 6.6: Crowd voting results

Measure	**Number**
All votes	8,611
Votes over unique asserts	5,436
Yes votes	7,560
No votes	1,051
Rejected answers	636
Accepted answers	4,703
Undecided answers	97

If we look at some of the most rejected assertions, we can see assertions such as (more no votes than yes):

- $nationalLanguage(DenmarkBornholmsk)$ (3 yes, 9 no)
- $typicalColorOfType(AutomobileBlackColor)$ (3 yes, 9 no)

- $is(CityOfCopenhagenDenmarkTown)$ (0 yes, 4 no)
- $subclass(Coffee - BeverageArtificialMaterial)$ (3 yes, 9 no)
- $soleMakerOfProductType(TelevisionSetPhilipsPetroleumCompany)$ (3 yes, 7 no)

And the most accepted assertions (more yes votes than no):

- $is(DenmarkCountryWithOnlyOneTimeZone)$ (25 yes, 0 no)
- $is(FoodCollectionWithManySpecializzations)$ (23 yes, 0 no)
- $sublcass(FoodOrganicMaterial)$ (19 yes, 0 no)
- $is(FoodProductTypeWithoutSoleMaker)$ (20 yes, 1 no)
- $countryPhoneCode(Slovenia386)$ (18 yes, 0 no)

And some of the undecided assertions (same number of yes and no votes):

- $subclass(IceCreamMixture)$ (2 yes, 2 no)
- $characteristicProperSubeventTypes(CookingFoodEatingEvent)$ (1 yes, 1 no)
- $electronicDeviceMountingStyle(RearVideoPortDesktopComputer)$ (2 yes, 2 no)

6.1.5 User Stickiness Factor

To get an idea of how the users interacted with the system, we checked each of them by the first and the last piece of knowledge they provided and measured the duration of days between these dates. This is a simple indicator of a stickiness - showing how many of the users only tried an application and then forgot about it and how many of them answered questions for longer. On Figure 6.3 we can see the long-tail distribution of how long users stick with the application. About 25% of the users stick with the system for more than just one-day testing. While most of the users (548) only tested the system for a day and did not use it further, 180 users stayed for a longer period of time. Among these, 117 used it for more than a week, 98 for more than two weeks, 69 for a month or more, 51 for more than two months and 15 for more than a year.

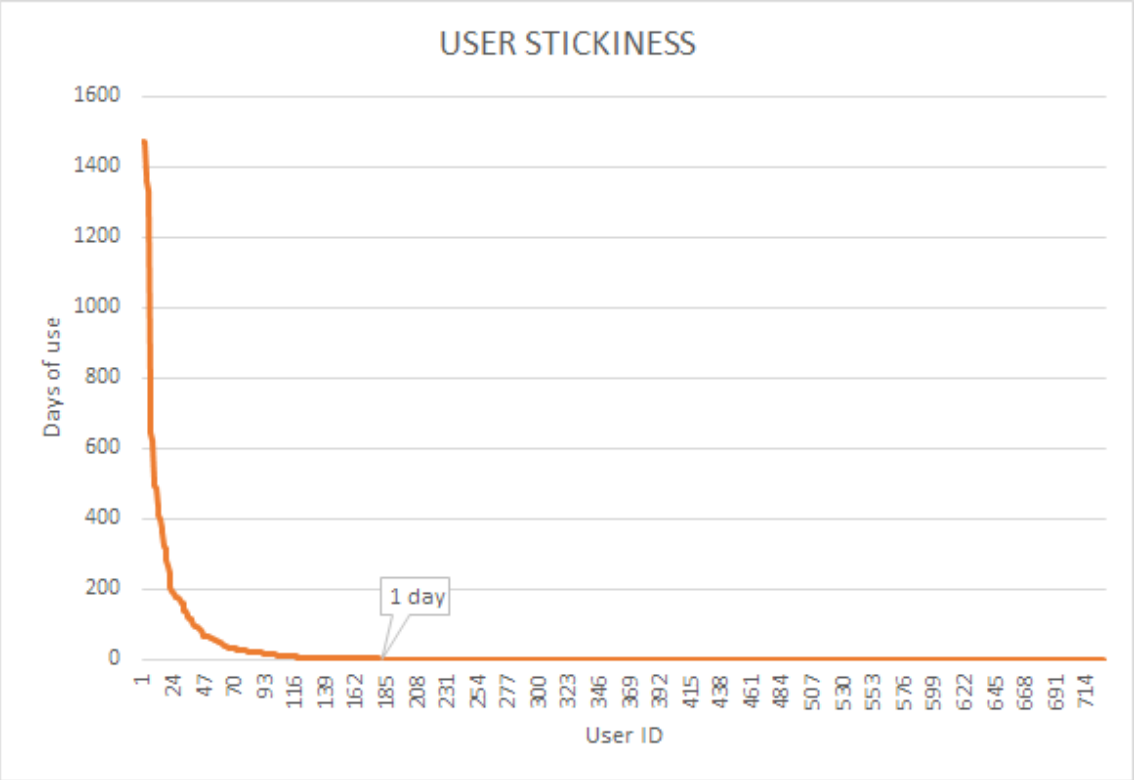

Figure 6.3: Distribution of usage duration for users

While this distribution is not too impressive for the commercial product, it is somehow expected for a research prototype without any updates and improvements for more than a year after it was built. It shows that the approach has potential and triggered interest in a decent number of the users, which indicates it could gain a lot of interest with a few improvements and obvious bug fixes.

6.1.6 Results Summary

As a wrap-up of the result sections, Table 6.7 shows the knowledge or quality contributions of particular system features and their appropriate baselines.

As the overall sanity check of the KB, we randomly picked 100 newly acquired assertions, and assessed them manually to see whether they were:

- Valid (in the sense of consistent with the KB here we expect 100%)
- True (in the sense of true in an interpretation based on our human world)
- Useful (useful for a potential user, or for the inference engine in producing suggestions, validations, new questions)

The counts are presented in Table 6.8. As expected all the assertions are valid, 96% of them are true and 95% are useful. This can be used as an estimate of the knowledge that we acquired through the proposed approach.

The results of this counting are not surprising, since the system does not allow manifestly inconsistent assertions, and the crowdsourcing mechanism already weeded out most of the untrue examples. There were four untrue assertions in the sample, which expose three potential problems:

Table 6.7: Contributions of separate system features and appropriate baselines

System Feature	Measure	Baseline	Proposed Approach
Knowledge Gathering			
Context and Proactivity	Number of gathered assertions/day	71	**42.1**
Follow-up assertions (new knowledge due to already collected k.	Number of gathered assertions	35,140	**57,978**
	Number of gathered assertions per concept	0	**4.22**
Cross-user assertions	Number of gathered assertions (extrapolated)	43,920	**57,978**
	Number of gathered assertions/concept	0	**2.73**
Quality Control			
Consistency Checking	Number of removed bad answers	0	**1420**
Crowd Voting	Number of removed bad answers	0	**733**

Table 6.8: Results of manual evaluation on 100 randomly picked assertions

Valid	True	Useful
100	96	95

1. One error was because the price of the coffee changed since the last check with the users the stored knowledge was obsolete.

2. There were two concepts with exactly the same name and were both subclasses of *Organization*, so the users and also the system did not manage to distinguish them and picked the wrong one. For users, everything seemed perfectly correct even though a logical error resulted.

3. Twice the user entered a complex sentence instead of a name of the concept and that forced the system to create a concept with that name (a consequence of disabled SCG (see subsection 5.1.3).

Looking more in details of the usefulness of the retrieved knowledge, there were two answers, which related to the internal KB mechanism (how to handle events) and were thus not really useful for the end user. The remaining there were mostly too specific and not really useful in general, such as the name of the spider the user has in the corner of a room.

Chapter 7

Conclusions

This thesis proposes a novel novel mobile, context aware, conversational crowdsourcing knowledge acquisition approach which is able to gather new knowledge of a very high quality in a continuous way, while interacting with its users using natural language. This is achieved by using an existing knowledge base, users current and past context, and employing a targeted crowdsourcing methodology. We proposed to use highly focused context to support targeting the right users at the right time. This had (to our knowledge) not been tried before. The proposed approach was implemented as part of Curious Cat system and evaluated on the the data gathered during the experiment running online throughout four years.

The results of the experiment showed that the approach is feasible and gives a good quality results as knowledge that is immediately useful, even if the users (crowd) does not have a knowledge engineering background. It also shows that it is possible to incorporate KA into the natural language HCI interaction and thus gather knowledge as a side effect of using the application for some other purposes. The results also confirm that all of the main features of the approach are contributing to the quality and quantity of the collected data, while making users interested enough (25% using the app for more than a day, 2% for more than a year), besides being in a research prototype state. The context and proactivity increase engagement from 7.1 assertions/day to 42.1/day. Using newly acquired knowledge to produce more and better questions acquired 39.4% of all the knowledge. On the other side consistency checking improved the knowledge base by preventing bad assertions (2.41% of all user answers) and crowd voting prevented consistent but otherwise wrong or untrue assertions (1.26% of all user answers).

7.1 Future Work

There are several potential directions for the future work, depending which part of the system we want to improve first. To improve the user engagement (besides the obvious one to improve and finish the system or make it part of some really useful application), is to apply machine learning on top of using the only inference rules. Since the system already collects various features for when and how much users answer, we could try to apply some ranking algorithms to improve the ordering of the questions (at the cases when we have multiple of them for the same concept), and thus either optimize the knowledge gain, or user interest, or both if possible.

The other direction is to extend the system to be able to acquire also the predicates and the rules, by finding a ways to ask questions which combine multiple concepts in a new ways and let them get confirmed by the users. Another direction is to analyze the free text questions and answers that users give, especially through MMHHI (subsubsection 4.1.1.3),

and mine the patterns, rules and predicates from these, then use the described KA approach to verify and confirm the mined knowledge.

One of the other promising directions for the future work is to improve the approach to become a full conversational engine (or to be merged with some existing one), which would make it easy to construct knowledgeable chat-bots. This would allow improvement of the system to accept and understand more complex answers and questions from the users, and to not limit itself only to short replies and guided responses. This is to some extent similar to the current AIML and ChatScript systems, except that *Curious Cat* is completely knowledge driven and designed to be able to extend itself indefinitely while talking with the users, and is only using patterns for NL to knowledge conversion (as opposed to the existing approaches).

References

Ahn, L. von (2006). "Games with a Purpose." In: *Computer* 39.6, pp. 92–94. ISSN: 0018-9162. DOI: 10.1109/MC.2006.196. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1642623.

Ahn, Luis Von, Mihir Kedia, and Manuel Blum (2006). "Verbosity : A Game for Collecting Common-Sense Facts." In: *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pp. 75–78. ISBN: 1595931783.

Ahn, Luis von and Laura Dabbish (2008). "Designing games with a purpose." In: *Communications of the ACM* 51.8, p. 57. ISSN: 00010782. DOI: 10.1145/1378704.1378719.

Baral, Chitta, J. Dzifcak, and L. Tari (2007). "Towards overcoming the knowledge acquisition bottleneck in answer set prolog applications: Embracing natural language inputs." In: *International Conference on Logic Programming (ICLP)* 4670 LNCS, pp. 1–21. ISSN: 03029743 16113349.

Baxter, David et al. (2005). "Interactive Natural Language Explanations of Cyc Inferences." In: *In AAAI 2005: International Symposium on Explanation-aware Computing*.

Bernstein, Michael et al. (2009). "Collabio: a game for annotating people within social networks." In: *Proceedings of the 22nd annual ACM symposium on User interface software and technology (UIST '09)*, pp. 97–100. ISSN: 00325910. DOI: 10.1145/1622176.1622195. URL: http://dl.acm.org/citation.cfm?id=1622195.

— (2010). "Personalization via friendsourcing." In: *ACM Transactions on Computer-Human Interaction* 17.2, pp. 1–28. ISSN: 10730516. DOI: 10.1145/1746259.1746260.

Bollacker, Kurt et al. (2008). "Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge." In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. New York, NY, USA: ACM, pp. 1247–1250. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376746. URL: http://doi.acm.org/10.1145/1376616.1376746.

Bradeško, Luka and Dunja Mladenić (2012). "A Survey of Chabot Systems through a Loebner Prize Competition." In: *Proceedings of Slovenian Language Technologies Society Eighth Conference of Language Technologies*, pp. 34–37. ISBN: ISBN 978-961-264-048-4.

Bradeško, Luka, Alexandra Moraru, et al. (2012). "A framework for acquiring semantic sensor descriptions (short paper)." In: *CEUR Workshop Proceedings* 904, pp. 97–102. ISSN: 16130073. URL: http://sensorlab.ijs.si/files/publications/2012-Bradesko-A%7B%5C_%7DFramework%7B%5C_%7Dfor%7B%5C_%7DAcquiring%7B%5C_%7DSemantic%7B%5C_%7DSensor%7B%5C_%7DDescriptions.pdf.

Bradesko, Luka et al. (2015). *Integrated Mobility Decision Support (Mobis D4.5)*. Tech. rep. EU Project deliverable (MOBIS).

Cambria, Erik et al. (2015). "Game Engine for Commonsense Knowledge Acquisition." In: *The Twenty-Eighth International Flairs Conference*, pp. 282–287.

Coppock, Elizabeth and David Baxter (2010). "A translation from logic to english with dynamic semantics." In: *Lecture Notes in Computer Science (including subseries Lec-*

ture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 6284 LNAI, pp. 197–216. ISSN: 03029743. DOI: 10.1007/978-3-642-14888-0_18.

Costa, Ruben et al. (2016). "Personalized Intelligent Mobility Platform: An Enrichment Approach Using Social Media." In: *Intelligent Decision Technology Support in Practice*. Ed. by Jeffrey W. Tweedale et al. Springer International Publishing. Chap. 5, pp. 61–87. ISBN: 978-3-319-21208-1. DOI: 10.1007/978-3-319-21209-8. URL: http://www.springer.com/gp/book/9783319212081.

Coursey, Kino (2004). "LIVING IN CYN : MATING AIML AND CYC TOGETHER WITH PROGRAM N." In:

Demner-Fushman, Dina et al. (2015). "YAGO3: A Knowledge Base from Multilingual Wikipedias." In: *Conference on Innovative Data Systems Research (CIDR)*. URL: http://scholar.google.com/scholar?hl=en%7B%5C&%7DbtnG=Search%7B%5C&%7Dq=intitle:Automatic+Event+and+Relation+Detection+with+Seeds+of+Varying+Complexity%7B%5C#%7D0%7B%5C%%7D5Cnhttp://www.aclweb.org/anthology/C12-1129%7B%5C%%7D5Cnhttp://dx.doi.org/10.1016/j.jbi.2013.08.010%7B%5C%%7D5Cnhttp://www.informatik.uni.

Dong, Zhendong, Qiang Dong, and Changling Hao (2010). "HowNet and Its Computation of Meaning." In: *Coling 2010* August, pp. 53–56. DOI: 10.1142/9789812774675.

Downey, Doug, Oren Etzioni, and Stephen Soderland (2005). "A probabilistic model of redundancy in information extraction." In: *IJCAI International Joint Conference on Artificial Intelligence*, pp. 1034–1041. ISSN: 10450823. DOI: 10.1016/j.artint.2010.04.024.

Eslick, Ian Scott (2006). "Searching for Commonsense." Doctoral dissertation.

Etzioni, Oren, Anthony Fader, et al. (2011). "Open Information Extraction: The Second Generation." In: *Proc. Int. Joint Conf. Artificial Intell.*

Etzioni, Oren, Ana-maria Popescu, et al. (2004). "Web-Scale Information Extraction in KnowItAll (Preliminary Results)." In: *WWW 2004*.

Fader, Anthony, Stephen Soderland, and Oren Etzioni (2011). "Identifying relations for open information extraction." In: *Proceedings of the Conference on ...* pp. 1535–1545. ISSN: 1937284115. DOI: 10.1234/12345678. arXiv: arXiv:1411.4166v4. URL: http://dl.acm.org/citation.cfm?id=2145596%7B%5C%%7D5Cnhttp://dl.acm.org/citation.cfm?id=2145596%7B%5C%%7D5Cnhttp://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.226.1089%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf%7B%5C%%7D5Cnhttp://www.cs.washington.edu/research/projects/aiweb/media/papers/etzioni-ijca.

Feigenbaum, E. A. (1977). "The Art of Artificial Intelligence: Themses and Case Studies of Knowledge Engineering." In: *Proceedings of the 5th International Joint Conference of Artificial itelligence*, pp. 1014–1029.

Figueiras, Paulo et al. (2013). "Knowledge base approach for developing a mobile personalized travel companion." In: *2013 13th International Conference on ITS Telecommunications, ITST 2013*, pp. 97–103. DOI: 10.1109/ITST.2013.6685528.

Forbus, Kenneth D et al. (2007). "Integrating Natural Language , Knowledge Representation and Reasoning , and Analogical Processing to Learn by Reading Learning Reader : The System." In: *Proceedings of AAAI-07: Twenty-Second Conference on Artificial Intelligence*. Vancouver,BC.

Geller, Tom (2016). "Can Chatbots Think Before They Talk?" In: *Communication of the ACM*. URL: https://cacm.acm.org/news/201579-can-chatbots-think-before-they-talk/fulltext.

Hasbro (n.d.). *Taboo board game*. URL: https://www.hasbro.com/common/documents/dad288731c4311ddbd0b0800200c9a66/2BF862075056900B1021F6D7061EDCC7.pdf.

Johan de Kleer (2013). *Method and apparatus for maintaining groupings and relationships of propositions associated with a knowledge base.* URL: https://encrypted.google.com/patents/US8401988.

J.R., Quinlan and Cameron-Jones R.M. (1997). "Induction of Logic Programs: FOIL and related systems." In: *New Generation Computing* 13.3, pp. 287–312. DOI: 10.1007/BF03037228. URL: https://doi.org/10.1007/BF03037228.

Kang, Jong Hee et al. (2005). "Extracting places from traces of locations." In: *ACM SIGMOBILE Mobile Computing and Communications Review* 9.3, p. 58. ISSN: 15591662. DOI: 10.1145/1094549.1094558.

Kazic, Blaz, Luka Bradesko, and Jan Rupnik (2017). "Predicting Users' Mobility Using Monte Carlo Simulations." In: *IEEE Access*.

Kuo, Yen-Ling and Jane Yung-jen Hsu (2010). "Goal-Oriented Knowledge Collection." In: *AAAI Fall Symposium: Commonsense Knowledge*, pp. 64–69. URL: http://www.aaai.org/ocs/index.php/FSS/FSS10/paper/viewPDFInterstitial/2278/2605.

Kuo, Yen-ling et al. (2009). "Community-Based Game Design: Experiments on Social Games for Commonsense Data Collection." In: *Proceedings of the ACM SIGKDD Workshop on Human Computation (HCOMP)*, pp. 15–22. ISSN: 978-1-60558-193-4. DOI: 10.1145/1600150.1600154. URL: http://dl.acm.org/citation.cfm?id=1600150.1600154.

Lehmann, Jens et al. (2015). "DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia." In: *Semantic Web* 6.2, pp. 167–195. ISSN: 22104968. DOI: 10.3233/SW-140134.

Lenat, Douglas B., Mayank Prakash, and Mary Shepherd (1985). "CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlenecks." In: *AI Magazine* 6.4, p. 65. ISSN: 0738-4602. DOI: 10.1609/aimag.v6i4.510. URL: http://www.aaai.org/ojs/index.php/aimagazine/article/view/510.

Lenat, Douglas Bruce (1995). "Cyc: A Large-Scale Investment in Knowledge Infrastructure." In: *Communications of the ACM* 38.22.

Luka, Bradesko et al. (2016). "Conversational Crowd based and Context Aware Knowledge Acquisition Chat Bot (Best Paper Award)." In: *8th IEEE International Conference on Intelligent Systems IS'16*, pp. 239–252. ISBN: 9781509013548.

Lv, Mingqi et al. (2016). "The discovery of personally semantic places based on trajectory data mining." In: *Neurocomputing* 173, pp. 1142–1153. ISSN: 18728286. DOI: 10.1016/j.neucom.2015.08.071. URL: http://dx.doi.org/10.1016/j.neucom.2015.08.071.

Mamei, Marco (2010). "Applying Commonsense Reasoning to Place Identification." In: *International Journal of Handheld Computing Research* 1.2, pp. 36–53. DOI: 10.4018/jhcr.2010040103. URL: http://dx.doi.org/10.4018/jhcr.2010040103.

Martin, E and I Riesbeck (1986). "Uniform Parsing and Inferencing for Learning." In: *Proceedings of AAAI-86*, pp. 257–261.

Masters, James, Cynthia Matuszek, and Michael Witbrock (2007). "Ontology-Based Integration of Knowledge from Semi-Structured Web Pages." In: *Cycorp*.

Matuszek, Cynthia, John Cabral, et al. (2006). "An Introduction to the Syntax and Content of Cyc." In: *Proceedings of the 2006 AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*. AAAI Press.

Matuszek, Cynthia, Michael Witbrock, et al. (2004). "Searching for Common Sense : Populating Cyc ™ from the Web." In: *Search*.

McKinstry, Chris, Rick Dale, and Michael J. Spivey (2008). "Action dynamics reveal parallel competition in decision making." In: *Psychological Science* 19.1, pp. 22–24. DOI: 10.1111/j.1467-9280.2008.02041.x.

Medelyan, Olena and Catherine Legg (2008). "Integrating Cyc and Wikipedia: Folksonomy meets rigorously defined common-sense." In: *Proceedings of the WIKIAI Wikipedia and AI Workshop at the AAAI* 8, pp. 13–18. URL: http://www.aaai.org/Papers/Workshops/2008/WS-08-15/WS08-15-003.pdf.

Mitchell, T et al. (2015). "Never-Ending Learning." In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI-15)*.

Mueller, Erik T (1999). *A database and lexicon of scripts for ThoughtTreasure*. Vol. 1999. CogPrints ID cog00000555 http://cogprints.soton.ac.uk, Article No. 0003004.

— (2003). *ThoughtTreasure: A natural language/commonsense platform*. URL: http://alumni.media.mit.edu/%7B~%7Dmueller/papers/tt.html (visited on 01/01/2017).

Otani, Naoki et al. (2016). "No TitleLarge-Scale Acquisition of Commonsense Knowledge via a Quiz Game on a Dialogue System." In: *Proceedings of the Open Knowledge Base and Question Answering (OKBQA) Workshop*. Osaka, pp. 11–20. ISBN: 9784879747129. URL: https://pdfs.semanticscholar.org/c20e/a24fed973aa6c5d70b86d5b22b4bd2e18811.pdf%7B%5C#%7Dpage=23.

Panton, Kathy et al. (2002). "Knowledge Formation and Dialogue Using the KRAKEN Toolset." In: *Proceedings of the Fourteenth National Conference on Innovative Applications of Artificial Intelligence*, pp. 900–905.

Pedro, S D S, A P Appel, and E R Hruschka Jr (2013). "Autonomously reviewing and validating the knowledge base of a never-ending learning system." In: *Proceedings of the 22nd ...* pp. 1195–1203. ISBN: 9781450320382. URL: http://dl.acm.org/citation.cfm?id=2488149.

Pedro, Saulo D. S. and Estevam R. Hruschka (2012). "Collective intelligence as a source for machine learning self-supervision." In: *Proceedings of the 4th International Workshop on Web Intelligence & Communities - WI&C '12*. 3, p. 1. ISBN: 9781450311892. DOI: 10.1145/2189736.2189744. URL: http://dl.acm.org/citation.cfm?id=2189736.2189744.

Rebele, T. et al. (2016). "YAGO: A Multilingual Knowledge Base from Wikipedia, Wordnet, and Geonames." In: *Groth P. et al. (eds) The Semantic Web – ISWC 2016*. Cham: Springer. DOI: https://doi.org/10.1007/978-3-319-46547-0_19. URL: https://link.springer.com/chapter/10.1007/978-3-319-46547-0%7B%5C_%7D19%7B%5C#%7Dciteas.

Schneider, Dave and Michael Witbrock (2015). "Semantic Construction Grammar : Bridging the NL / Logic Divide." In: pp. 673–678.

Schubert, Lenhart (2002). "Can we derive general world knowledge from texts?" In: *Proceedings of the second international conference on Human Language Technology Research*, p. 94. DOI: 10.3115/1289189.1289263. URL: http://portal.acm.org/citation.cfm?doid=1289189.1289263.

Schubert, Lenhart and Matthew Tong (2003). "Extracting and evaluating general world knowledge from the Brown corpus." In: *Proceedings of the HLT-NAACL 2003 workshop on Text meaning - Volume 9*, pp. 7–13. DOI: 10.3115/1119239.1119241. URL: http://dx.doi.org/10.3115/1119239.1119241.

Sharma, Abhishek and KD Forbus (2010). "Graph-Based Reasoning and Reinforcement Learning for Improving Q/A Performance in Large Knowledge-Based Systems." In: *2010 AAAI Fall Symposium Series*, pp. 96–101. ISBN: 9781577354840. URL: http://www.aaai.org/ocs/index.php/FSS/FSS10/paper/download/2246/2596.

Singh, Push (2002). "The Public Acquisition of Commonsense Knowledge Push Singh The Diversity of Commonsense Knowledge." In: *AAAI Spring Symposium: Acquiring (and Using) Linguistic (and World) Knowledge for Information Access*, pp. 47–53. URL:

http://www.aaai.org/Papers/Symposia/Spring/2002/SS-02-09/SS02-09-011.pdf.

Singh, Push et al. (2002). "Open Mind Common Sense: Knowledge acquisition from the general public." In: *Cooperative Information Systems Oct. 30-Nov. 1 2002*, pp. 1223–1237. ISSN: 03029743. DOI: 10.1007/3-540-36124-3_77. URL: http://portal.acm.org/citation.cfm?id=646748.701499.

Soderland, Stephen et al. (2007). "Open information extraction from the web." In: *International Joint Conference On Artificial Intelligence*, pp. 2670–2676. ISSN: 00010782. DOI: 10.1145/1409360.1409378. URL: http://portal.acm.org/citation.cfm?id=1625705.

Speer, Robert (2007). "Open mind commons: An inquisitive approach to learning common sense." In: *Proceedings of the Workshop on Common Sense and Interactive Applications*. URL: http://www.fatih.edu.tr/%7B~%7Dhugur/inquisitive/Open%20Mind%20Commons%20An%20Inquisitive%20Approach%20to.PDF.

Speer, Robert, Joshua Chin, and Catherine Havasi (2016). "ConceptNet 5.5: An Open Multilingual Graph of General Knowledge." In: Singh 2002. arXiv: 1612.03975. URL: http://arxiv.org/abs/1612.03975.

Speer, Robert, Jayant Krishnamurthy, et al. (2009). "An interface for targeted collection of common sense knowledge using a mixture model." In: *Proceedings of the 14th International Conference on Intelligent User Interfaces*, pp. 137–146. DOI: 10.1145/1502650.1502672.

Speer, Robert, Henry Lieberman, and Catherine Havasi (2008). "AnalogySpace : Reducing the Dimensionality of Common Sense Knowledge." In: *AAAI'08 Proceedings of the 23rd national conference on Artificial intelligence*, pp. 548–553. ISBN: 9781577353683.

Suchanek, Fabian M., Gjergji Kasneci, and Gerhard Weikum (2008). "YAGO: A Large Ontology from Wikipedia and WordNet." In: *Web Semantics* 6.3, pp. 203–217. ISSN: 15708268. DOI: 10.1016/j.websem.2008.06.001. arXiv: arXiv:1203.5073v1.

Wagner, Christian (2006). "Breaking the Knowledge Acquisition Bottleneck Through Conversational Knowledge Management." In: *Information Resources Management Journal* 19.1, pp. 70–83. ISSN: 1040-1628. DOI: 10.4018/irmj.2006010104. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.3138%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

Wallace, Richard (2013). "AIML 2.0 Draft Specification." URL: http://www.alicebot.org/style.pdf.

Wallace, Richard S. (2003). *The Elements of AIML Style*. Tech. rep. Alice AI Foundation. DOI: 10.1.1.693.3664. URL: http://www.alicebot.org/style.pdf.

Weizenbaum, Joseph (1966). "ELIZA–A Computer Program For the Study of Natural Language Communication Between Man and Machine." In: *Communication of the ACM* 9.1, pp. 36–45. ISSN: 00010782. DOI: 10.1145/365153.365168.

Wilcox, Bruce (2011). "Beyond Façade: Pattern Matching for Natural Language Applications." In: *Gamasutra*, pp. 1–5. URL: http://www.gamasutra.com/view/feature/6305/beyond%7B%5C_%7Dfa%EF%BF%BDade%7B%5C_%7Dpattern%7B%5C_%7Dmatching%7B%5C_%7D.php?page=1.

Witbrock, Michael (2010). "Acquiring and Using Large Scale Knowledge Knowledge Capture : Mixed Initiative." In: *Proceedings of the ITI 2010, 32nd International Conference on Information Technology Interfaces*. Cavtat/Dubrovnik, pp. 37–42. URL: http://ieeexplore.ieee.org/document/5546360/?reload=true%7B%5C&%7Dtp=%7B%5C&%7Darnumber=5546360.

Witbrock, Michael, David Baxter, et al. (2003). "An Interactive Dialogue System for Knowledge Acquisition in Cyc." In: *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*. Acapulco, Mexico.

Witbrock, Michael and Luka Bradesko (2013). "Conversational Computation." In: *Handbook of Human Computation*. Ed. by Pietro Michelucci. Springer, pp. 531–543. ISBN: 978-1-4614-8806-4. DOI: https://doi.org/10.1007/978-1-4614-8806-4_40. URL: https://doi.org/10.1007/978-1-4614-8806-4%7B%5C_%7D40.

Witbrock, Michael, Cynthia Matuszek, et al. (2005). "Knowledge Begets Knowledge: Steps towards Assisted Knowledge Acquisition in Cyc." In: *AAAI Spring Symposium: Knowledge Collection from Volunteer Contributors*, pp. 99–105. URL: http://www.aaai.org/Papers/Symposia/Spring/2005/SS-05-03/SS05-03-015.pdf.

Wu, Wentao et al. (2012). "Probase: A probabilistic taxonomy for text understanding." In: *Proceedings of the 2012 ACM SIGMOD ...* pp. 481–492. ISSN: 00043702. DOI: 10.1016/j.artint.2011.01.003. URL: http://dl.acm.org/citation.cfm?id=2213891.

Zang, Liang-Jun et al. (2013). "A Survey of Commonsense Knowledge Acquisition." In: *Journal of Computer Science and Technology* 28.4, pp. 689–719. ISSN: 1000-9000. DOI: 10.1007/s11390-013-1369-6. URL: http://link.springer.com/10.1007/s11390-013-1369-6.

Bibliography

Publications Related to the Thesis

All publications related to the thesis should be referenced in the text.

Journal Articles

Bradesko, Luka, Michael Witbrock, et al. (2017). "Curious Cat-Mobile, Context-Aware Conversational Crowdsourcing Knowledge Acquisition." In: *ACM Trans. Inf. Syst.* 35.4, 33:1–33:46. DOI: 10.1145/3086686. URL: http://doi.acm.org/10.1145/3086686.

Kazic, Blaz, Luka Bradesko, and Jan Rupnik (2017). "Predicting Users' Mobility Using Monte Carlo Simulations." In: *IEEE Access*.

Conference Paper

Bradeško, Luka and Dunja Mladenić (2012). "A Survey of Chabot Systems through a Loebner Prize Competition." In: *Proceedings of Slovenian Language Technologies Society Eighth Conference of Language Technologies*, pp. 34–37. ISBN: ISBN 978-961-264-048-4.

Bradeško, Luka, Alexandra Moraru, et al. (2012). "A framework for acquiring semantic sensor descriptions (short paper)." In: *CEUR Workshop Proceedings* 904, pp. 97–102. ISSN: 16130073. URL: http://sensorlab.ijs.si/files/publications/2012-Bradesko-A%7B%5C_%7DFramework%7B%5C_%7Dfor%7B%5C_%7DAcquiring%7B%5C_%7DSemantic%7B%5C_%7DSensor%7B%5C_%7DDescriptions.pdf.

Figueiras, Paulo et al. (2013). "Knowledge base approach for developing a mobile personalized travel companion." In: *2013 13th International Conference on ITS Telecommunications, ITST 2013*, pp. 97–103. DOI: 10.1109/ITST.2013.6685528.

Luka, Bradesko et al. (2016). "Conversational Crowd based and Context Aware Knowledge Acquisition Chat Bot (Best Paper Award)." In: *8th IEEE International Conference on Intelligent Systems IS'16*, pp. 239–252. ISBN: 9781509013548.

Book Chapter

Costa, Ruben et al. (2016). "Personalized Intelligent Mobility Platform: An Enrichment Approach Using Social Media." In: *Intelligent Decision Technology Support in Practice*. Ed. by Jeffrey W. Tweedale et al. Springer International Publishing. Chap. 5, pp. 61–87. ISBN: 978-3-319-21208-1. DOI: 10.1007/978-3-319-21209-8. URL: http://www.springer.com/gp/book/9783319212081.

Witbrock, Michael and Luka Bradesko (2013). "Conversational Computation." In: *Handbook of Human Computation*. Ed. by Pietro Michelucci. Springer, pp. 531–543. ISBN: 978-1-4614-8806-4. DOI: https://doi.org/10.1007/978-1-4614-8806-4_40. URL: https://doi.org/10.1007/978-1-4614-8806-4%7B%5C_%7D40.

Magazine Article

Geller, Tom (2016). "Can Chatbots Think Before They Talk?" In: *Communication of the ACM*. URL: https://cacm.acm.org/news/201579-can-chatbots-think-before-they-talk/fulltext.

Other Publications (optional)

...

Biography

The author of this thesis is PhD candidate from Artificial Intelligence Lab at Jozef Stefan Institute, Slovenia. His research interests and PhD topic are in Natural Language Processing, Logical Inference, Knowledge Extraction and Crowdsourcing. In 2010 he graduated from Faculty of Electrical Engineering and started his doctoral studies at Jožef Stefan International Postgraduate School. He worked as an Artificial Intelligence and Machine Learning researcher at Jožef Stefan Artificial Intelligence Laboratory since 2005 on the topics related to this thesis. From 2008 to 2013 he also worked as a principal software engineer for Cycorp Europe, which was at the time an EU branch of the American AI company Cyc Inc. During these years, he worked on an EU project developing distributed large scale inference engine (LarKC), and also on an AI assistant build on top of Cyc which lead to the development and implementation of the approach described in this thesis. Some of the recent projects with a similar topic include a concept of an intelligent motorhome (reasoning engine software interacting with sensors and actuators) for a European motorhome producer (Adria Mobil) and Named Entity Disambiguation algorithm which is a work in progress in a collaboration with US Company Bloomberg L.P.