Curious Cat

Conversational Crowd based and Context Aware Knowledge Acquisition Chat Bot

Luka Bradeško, Janez Starc,
Dunja Mladenic, Marko Grobelnik
Artificial Intelligence Laboratory,
Jožef Stefan Institute,
Ljubljana, Slovenia,
luka.bradesko@ijs.si

Michael Witbrock
IBM Thomas J Watson Research Center,
Yorktown Heights, NY, USA,
witbrock@us.ibm.com

Abstract— **Acquisition of high quality structured knowledge that is immediately useful for reasoning algorithms has been a longstanding goal of the Artificial Intelligence research community. With the recent advances in crowdsourcing, the sheer number of internet users and the commercial availability of supporting platforms have come a new set of tools to tackle this problem. Although numerous systems and methods for crowdsourced knowledge acquisition had been developed and solve the problem of manpower, the issues of task preparation, financial cost, finding the right crowd, consistency, and quality of the acquired knowledge, seem to persist. In this paper we propose a new approach to address this deficit by exploiting an existing knowledge base to drive the acquisition process, address the right people, and check their answers for consistency. We conducted tests of the viability of the approach in experiments with real users, a working platform and common sense knowledge.**

Keywords-knowledge acquisition; crowdsourcing; reasoning; chatbots; knowledge systems; dialogue systems

I. Introduction

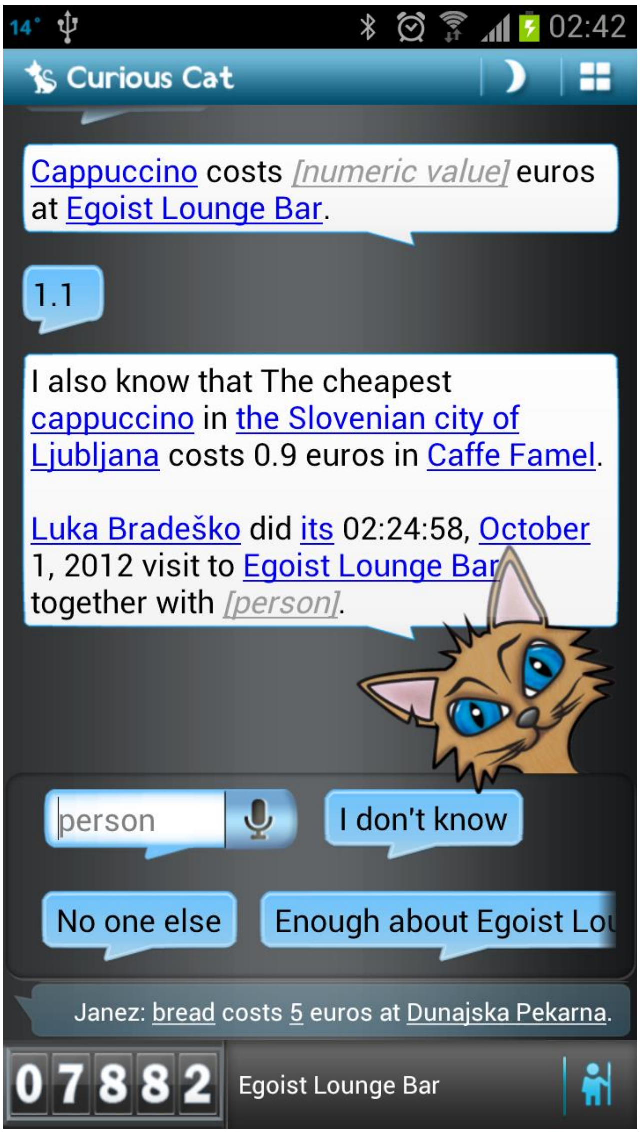
An intelligent being or machine solving any kind of a problem needs knowledge to which it can apply its intelligence in coming up with an appropriate solution. This is especially true for the knowledge-driven AI systems which constitute a significant fraction of general AI research. For these applications, getting and formalizing the right amount of knowledge is crucial. This knowledge is acquired by some sort of Knowledge Acquisition (KA) process, which can be manual, automatic or semi-automatic. Knowledge acquisition, using an appropriate representation and subsequent knowledge maintenance are two of the fundamental and as-yet unsolved challenges of AI. Knowledge is still expensive to retrieve and to maintain. This is becoming increasingly obvious, with the rise of chat-bots and other conversational agents and AI assistants. The most developed of these (Siri, Cortana, Google Now, Alexa), are backed by huge financial support from their producing companies, and the lesser-known ones still result from 7 or more person-years of effort by individuals (Wilcox's bots [1], etc.) or smaller companies (Josh AI). Besides machine learning and statistical approaches, a bigger chunk of effort in these systems lies in knowledge acquisition, which is sometimes hidden in (hand-)coded rules for request/response patterns and corresponding actions. This KA cost goes up linearly with the required knowledge coverage, which is much higher for general AI assistants compared to expert systems, which usually cover only specific task sets.

The novel approach to context-aware knowledge acquisition implemented in our system, Curious Cat (CC), enables reduction of the cost of the KA, in the long term, to the point where it will no longer be the bottleneck for building knowledge-dependent systems. This is achieved by:

- Not looking at KA as means to an end (expert system or AI agent), but as a part of the end itself, and thus incorporating KA into the end system as a part of its natural interaction.
- Building on top of existing knowledge and using it to understand the user's context and automatically drive the KA process further. This is done by automatically identifying missing pieces of knowledge and then deliberately trying to get the missing parts through the help of crowdsourcing and reasoning.

The main contribution of the paper is a novel knowledge acquisition approach that uses context and prior knowledge to automatically construct natural language crowdsourcing tasks for the right audience at the right moment. The newly acquired knowledge is then immediately used to construct better or more detailed questions and thus drive the KA process further. This makes our approach a self-maintained and ever growing natural language conversational agent that can be used for non-KA-related tasks as well.

The key idea is to use existing knowledge to ensure that the system at least partially "understands" a user's requests and its own responses, and to use that understanding, *inter alia*, to determine what it doesn't know, and to support asking for the missing knowledge. Answers to the resulting KA questions are then checked for consistency, validity and soundness against the existing knowledge. When there is not enough existing knowledge to be able to reject or support the user's answer, the system can ask other users, a form of on-demand crowdsourcing. In this way, Curious Cat is an almost self-sustained and self-maintained system, which performs its own KA in addition to the user-focused task it was designed for. Because its acquisition is highly controlled by the system, knowledge that can be acquired this way can be expected to be of very high quality, and is therefore automatically incorporated into the system's knowledge base without the need for paid review, resulting in cost savings.



I also know that The cheapest cappuccino in the Slovenian city of Ljubljana costs 0.9 euros in Caffe Famel.

Luka Bradeško did its 02:24:58, October 1, 2012 visit to Egoist Lounge Bar together with [person].

Janez: bread costs 5 euros at Dunajska Pekarna.

Figure 1: Screenshot of Curious Cat prototype

To test the approach and asses its usefulness, we built a proof-of-concept Conversational AI assistant (Figure 1) whose task is to tell users something interesting or useful about the places they visit, while being able to support incidental conversation ranging over all other common sense topics as well. This was achieved by building on top of Cyc [1], a very large pre-existing inferentially productive Common Sense Knowledge Base (KB), resulting from more than 900 (as of 2006 [2]) non-crowdsourced human years of effort. Besides the KB and the reasoning engine (again, Cyc in our implementation), the core components of the presented system are bidirectional natural language (NL) to logic conversion and a novel crowd-sourcing approach preserving user privacy and the ability to store the user's beliefs about the world. This means that the knowledge users provide through their answers, is independent of that from other users. It only affects other users when there is a need to check the truth, or when it is used to support asking something new. If a user deliberately or accidentally mislead the KA process, this would affect only the way the system interacts with him or her, with minimal impact on other users.

As a part of the experiment, Curious Cat has been publicly available online[1] since the end of 2012 and is still running after 3.5 years of continuous activity. In this time 600 users found the system and provided 394,060 pieces of knowledge. Curious Cat was recently also mentioned in Communications of the ACM[2].

Although this implementation used a specific knowledge base, reasoning engine and set of NLP methods, the approach is general and can be repeated using any sufficiently broad and inferentially productive existing KB and a reasoning engine with the appropriate meta-reasoning (reasoning about its own structures) capabilities.

II. Related Work

There are numerous approaches and systems that address KA from many perspectives, trying to improve its quality and reduce costs, with various levels of success. Some of the more successful approaches are described in the Common Sense KA survey [3], which organizes the approaches into four main groups:

1. **Labour Acquisition**. This approach uses human minds as the knowledge source. This usually involves expert ontologists coding the knowledge by hands.
2. **Interaction Acquisition**. As in Labour Acquisition, the source of the knowledge is human minds, but in this case the KA is wrapped in a facilitated interaction with the system, and is sometimes implicit rather than explicit.
3. **Reasoning Acquisition**. In this approach new knowledge is automatically inferred from the existing knowledge using logical rules and machine inference.
4. **Mining Acquisition**. In this approach, the knowledge is extracted from some large textual corpus or corpora.

Our approach is a hybrid system, addressing the problem via approaches 2 and 3 and adding the unique feature of using context and existing knowledge, with reasoning, to produce a practically unlimited number of potential Interaction Acquisition tasks. The existing related work can be divided into systems that exploit existing knowledge (generated anew during acquisition or pre-existing in other sources) [4, 5, 6, 7, 8, 9], crowdsourcing [8, 10, 11, 12, 9], acquisition through interaction [13, 11, 12], reasoning [14, 5, 15, 9], and natural language conversation [13, 14, 12, 5, 9].

One of the most similar systems (by functionality and approach) is Goal Oriented Knowledge Collection (GOKC) [9]. Like Curious Cat, GOKC starts with an initial seed KB

[1] https://play.google.com/store/apps/details?id=cc.curiouscat
[2] http://cacm.acm.org/news/201579-can-chatbots-think-before-they-talk/fulltext

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. For example, if we know a*tLocation* is linked with hasSubevent and we get an answer "class- room" to the question "You are likely to find in a school", we can ask players a new question "One thing you will do when you in classroom is __," [4]. 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 by voting.

Another related system, which is a predecessor of Curious Cat in some aspects, was the Cyc Kraken Systems' User Interaction Agenda (UIA) [5]. Like Curious Cat, the UIA was able to check answers for consistency, but it didn't use rules to explicitly drive question-asking. This feature was introduced in a new approach within Cyc: CURE (Content Understanding, Review and Entry, which is available within Research Cyc), which Curious Cat extends. In the UIA, the system relies on domain experts to select a concept, which then triggers a sequence of NL forms, which allow users to add new knowledge that may later be modified using "knowledge engineering tools". Both CURE and UIA are missing crowd-sourcing functionalities and pro-activity based on user context.

After an initial KB, extracted from internet textual content had been gathered, the CMU text-mining knowledge acquisition system NELL [4] (Never Ending Language Learner), started to apply a crowdsourcing approach [11], using natural language questions [13] 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, 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.

On the other end of the spectrum lie conversational agents (chatbots) with a vast number of hand-scripted NL patterns and responses. These patterns help the bot to appear intelligent, but in a limited way: the knowledge is only implicitly encoded in the patterns and cannot be used anywhere else. Recently, some successful chatbots have started to employ internal knowledge base as well. It is used to remember facts from the conversation for later use. In this way, the chat system is actually doing targeted knowledge acquisition. Two of the more successful approaches to Chatbot authoring are AIML [16] and ChatScript [17]; there has also been an attempt to extend AIML scripts with CYC knowledge [18], 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, Curious Cat generates NL responses based on knowledge and context driven inference. This allows our system to proactively 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 chatbots and actually understanding its content.

III. APPROACH

The primary goal of our conversational system is knowledge acquisition, with behavior as a conversation agent and assistant being secondary goals that serve as the means to drive KA. The aim is to perform KA effortlessly while having a conversation about concepts which have some connection to the user, allowing the system (or the user) to follow the links in the conversation to connected topics. This allows us to lead the conversation off topic for a while and possibly gather additional, unexpected knowledge, as can be seen in the example conversation sketch in Table I, where the topic changes from the specific restaurant, to the type of dish.

TABLE I DESIGN EXAMPLE OF A USER/CURIOUS CAT INTERACTION

Num.	**Interaction**	
	Curious Cat	***User1***
1	Where are we? Are we at L'Ardoise restaurant?	Yes
2	I've never been here before. What kind of cuisine do they serve?	French cuisine.
3	Thanks for teaching me that L'Ardoise serves French Food. Does L'Ardoise have baguettes on the menu?	Yes
…	Some time passes while user eats…	
4	What did you order?	Car
5	I've never heard of food called Car before. Are you sure it's a type of food?	No
6	What did you order then?	Traditional duck
7	I've never heard of 'traditional duck' before. What kind of thing is it?	Duck meat

All the knowledge gathered as part of the conversation is remembered by the system and is used to further generate new comments and related questions as shown in interaction number 3 from Table I. Additionally, the knowledge can be checked at any point with other users as in the example from Table II, below. Based on the votes (confirmations or rejections) from other users, the system can decide whether to believe the new knowledge in general, or only leave it in the user's own world theory. This is explained in more detail in section III.F (Crowdsourcing mechanisms).

TABLE II TRUTH CHECKING THROUGH THE HELP OF OTHER USERS

Num.	**Interaction**	
	Curious Cat	***User2***
8	Where are we? Are we at L'Ardoise restaurant?	Yes
9	I've heard about this place. Is it true that it serves French food?	Yes

As visible in the examples above, Curious Cat is knowledge driven, and uses an existing knowledge to be able to drive the conversation and ask better questions and thus gather even more knowledge. This is also its main differentiation from the other KA systems and conversational bots. It is not driven by some pre-defined scripts and textual patterns, but uses structured knowledge which can evolve and change through interaction and also independently through external context and sensors. Moreover, the KA process and the system are controllable [3] (and to some extent self-regulated) and actionable[4], which is not the case with statistical conversational client approaches such as, for example, one of the first - Cleverbot [19].

A. *General Architecture*

In Figure 2, we can see that the proposed system and its user interaction loop is built around the knowledge base (marked in purple), which stores the user context and world knowledge and also meta-knowledge such as KA rules, natural language representations of the concepts and rules for natural language generation. The knowledge base 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. Besides KA meta-knowledge, which we explain in section III.B.3), it is helpful if the KB already contains some prior knowledge which can be used for consistency checking.

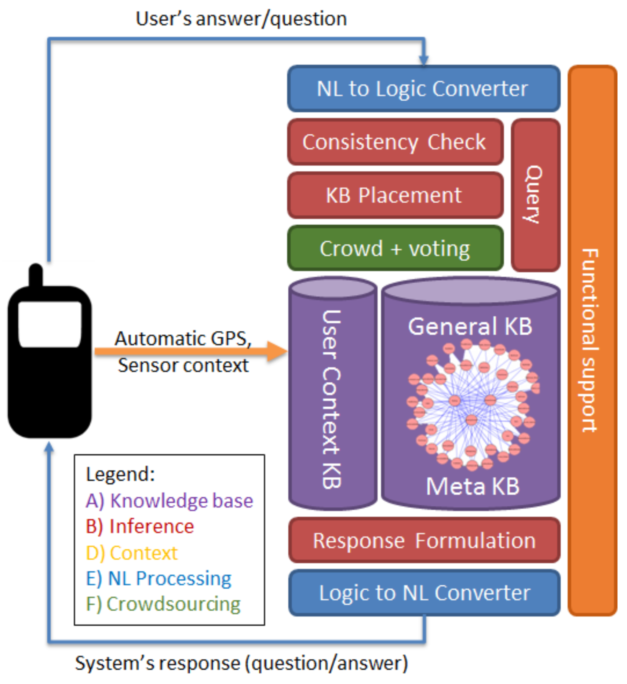


Figure 2. General Architecture of the KA system, with a simple interaction loop

[3] It is based on structured knowledge and it is thus possible for humans to review/understand and influence its behavioral rules.
[4] Because of the use of structured knowledge, it's possible to create special predicates representing actions in the external world, such us turning on the lights, or reserving a restaurant; not actionable in the legal sense.

Context representation and context sensitivity is an important feature of the system that allows us to target the right users at the right time and thus improve the efficiency and accuracy of the KA process. External context is provided by the mobile devices internet APIs and by external API's, and then asserted into the KB within the user's context.

The second most important part of the architecture is an inference engine (in Figure 2 represented in red), which is tightly connected to the knowledge base. The inference engine needs to be able to operate with the concepts, assertions and rules from the KB and should also be capable of meta-reasoning about the knowledge base's internal knowledge structures. The inference engine is used for:

- checking the consistency of the users' given answers (can you order a car in a restaurant if it's not food?),
- figuring out the placement of new knowledge inside the KB
- querying the KB to answer possible questions
- using knowledge and meta-rules to produce responses to user and context input (similar in function to the scripts in script-based conversational agents)

At both ends of the chain in Figure 2, we see natural language processing components (marked in blue), which are responsible for logic-to-language and language-to-logic conversion. These are crucial if we want to interact with users in a natural way and thus avoid the need for users to be experts in mathematical logic.

Besides the main interaction loop, which automatically uses crowdsourcing while it interacts with users, we have an additional component (marked in green). This "crowdsourcing and voting" component handles and decides, which elements of knowledge (logical assertions) can be safely asserted and made "visible" to all the users and which are questionable and should stay visible only to the authors of the knowledge. If the piece of knowledge is questionable, the system marks it as such and then the question formulation process will check with other users whether it's true or not. This is described in more detail in sub-section D (Crowdsourcing mechanisms).

In addition to logic-based components presented above, there is a functional driver system (marked in Orange), which glues everything together, forwards the results of inference to the NL converters, accepts and asserts the context into the KB, etc.

B. *Knowledge and KA Meta-knowledge*

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 only 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 I and Table II.

The examples are given in formal higher order predicate logic, which we later replace with a more compact notation, with a slight change in the way the variables are presented. For better readability, instead of *x*, *y*, *z*, we mark variables with a question mark (?) followed by a name that represents the expected type of the concepts the variable represents[5]. For example, when we see a variable in a logical formula like *CCUser*(?*PERSON)*, we immediately know that *?PERSON* can be replaced with instance or subclass of the concept *Person*. We start predicate names 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 [1], and while we use 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.[6]

1) Upper Ontology.

First we introduce the concepts that will allow us to present the upper ontology which is part of the knowledge base:

$$\textit{Something, Class, Predicate, subclass, is, arity, argClass, argIs} \tag{1}$$

Then we need to define what the relevant concepts are. In predicate logic, we can state things like:

$$Person(x) \tag{2}$$

Which means that *x* is a Person, or, more precisely, an instance of a class Person. We introduce a special predicate *is* that can tell us that something is an instance of something class, with the class being given as an argument:

$$\forall x \forall P\,(P(x) \Leftrightarrow is(x, P)) \tag{3}$$

Now, instead of *Person(x)* we can use the *is(x, Person)* notation, which will allow us to construct logical statements ranging over classes in a more transparent way.

Now we define the main predicates:

$$is(is, Predicate) \wedge is(subclass, Predicate) \wedge is(arity, Predicate) \wedge is(argClass, Predicate) \wedge is(argIs, Predicate) \tag{4}$$

From these definitions (4), we see that the *is* predicate can be used to define itself. In each of the assertions above, we used a concept *Predicate* for that class:

$$is(Predicate, Class) \wedge is(Class, Class) \wedge is(Something, Class) \tag{5}$$

Again, from the definition 5 above, we can see that Classes are instances of a concept *Class*, including *Class* itself. Predicates also have arity{ how many arguments the predicate can have. Let us define this for our predicates:

$$arity(is, 2) \wedge arity(subclass, 2) \wedge arity(arity, 2) \wedge arity(argClass, 3) \wedge arity(argIs, 3) \tag{6}$$

We can see now, that the arity of the *is* predicate is 2, and can confirm that all the logical formulas in the definitions above are correct, including those concerning *arity*; sa in previous examples, the *arity* predicate can be used to assert the arity of itself.

Then, because we want to be able to prevent our system from aquiring incorrect knowledge, we limit the domains of the arguments that can be used in each of the predicate. This could be done by entering specific material implication rules. For example, to limit the domains of both of *subclass* arguments, to be an instance of a *Class*, we could assert:

$$\forall x \forall y(subclass(x, y) \Rightarrow is(x, Class) \wedge is(y, Class)) \tag{7}$$

Because the rule is only true if the right part (the consequent) is true, or the left part (the antecedent) is false, its inclusion forces a KB to not allow the arguments of *subclass* to be anything else than an instance of a class *Class*. The rule above is rather complex and it would be hard to construct large knowledge base with numerous rules like this, which is why, following Cyc practice, we introduce the *argIs* predicate. Taken from the example above, we can expand our rule, to:

$$\forall x \forall y(((subclass(x, y) \Rightarrow (is(x, Class) \wedge is(y, Class))) \Leftrightarrow argIs(x, 1, Class) \wedge argIs(y, 2, Class)) \tag{8}$$

This rule (8) states that rule 7 can be written as 2 *argIs* assertions, which is much more readable. To make this hold for all the combinations of predicates, we can write a general rule, which will allow us to use *argIs* in all cases:

$$\forall x_{1..n} \forall P \forall C_{1..m} (((arity\,(P, n) \wedge P(x_1, \ldots, x_n) \Rightarrow (is(x_1, C_{1..m}) \wedge \ldots \wedge is(x_n, C_{1..m})))$$
$$\Leftrightarrow$$
$$(argIs(x_1, 1, C_{1..m}) \wedge \ldots \wedge argIs(x_n, n, C_{1..m})) \tag{9}$$

This now allows us to use *argIs* predicates instead of complicated rules. The predicate *argClass* can be defined similarly, except that it states that the argument must be a subclass of a specific Class., while *argIs* states that the argument must be instance of specific class. Let's now set, by assertion, the domain limits for our current knowledge base:

$$\begin{aligned} & argIs(argIs, 1, Predicate) \wedge argIs(argIs, 2, Number) \wedge \\ & argIs(argIs, 3, Class) \wedge argIs(argClass, 1, Predicate) \wedge \\ & argIs(argClass, 2, Number) \wedge argIs(argClass, 3, Class) \wedge \\ & argIs(is, 1, Something) \wedge argIs(is, 2, Class) \wedge \\ & argIs(subclass, 1, Class) \wedge argIs(subclass, 2, Class) \wedge \\ & argIs(arity, 1, Predicate) \wedge argIs(arity, 2, Number) \wedge \\ & argClass(argIs, 3, Something) \wedge \\ & argClass(argClass, 3, Something) \wedge \\ & argClass(is, 2, Something) \end{aligned} \tag{10}$$

We can see here, for example, that the argument 1 of arity must be instance of a *Predicate*, and its second argument must be a number. If we try to assert the arity of something that is not a predicate, we would make our KB inconsistent, and a

[5] The type here is merely notational; the system does not use type information from variable names. Types are enforced by the predicates, however.

[6] The notation here follows closely practices used in Cyc. For more details, readers can refer to [1] and [2] and the references it contains.

consistency check on assertion (as provided by Cyc) would block the change

Now, if we look closely, we see that our KB at this stage actually is inconsistent. The partial assertion above (definition 10101010): *argIs(is, 1, Something)* tells us that the first argument of predicate *is* must be of instance of *Something.* But in the definitions 4 and 5, we see that none of the first arguments used in *is* are instances of Something and also not all of the second arguments are instances of a *Class.* Let us first fix the *is(Something, Class)* assertion, so we can actually put *Something* into the first place of the *is* predicate. For this we need to introduce a rule, which states, that when x is an instance of y and y is simultaneously a subclass of z, then x is instance of z as well:

$$\forall x \forall y \forall z\ (is(x, y) \wedge subclass(y, z) \Rightarrow is\ (x, z)) \qquad (11)$$

Now, when we add the following two assertions:

$$subclass(Class, Something) \qquad (12)$$

Because of the rule 11, assertions in 12 and 5, all of our classes became instances of *Something* as well, and thus almost satisfy our knowledge base's domain constraints.

The remaining missing part is the constraint that the second arguments of *is* must be a subclass of *Something.* But this is not yet true for *Class* and *Predicate*. To solve this, we need to introduce two more rules. One to make the *subclass* predicate reflexive, and another one, to make it transitive:

$$\forall x subclass\ (x, x),$$
$$\forall x \forall y \forall z\ (subclass(x, y) \wedge subclass(y, z) \Rightarrow subclass\ (x, z)) \qquad (13)$$

Now *Predicate* and *Class* are both a subtypes of *Class* and thus our initial supporting KB is consistent.

At this time, we can also introduce a slightly easier to read notation, which will require less writing. For example, instead of enumerating all the assertions (arity, argIs, argClass, is Predicate, etc.) defining the subclass predicate, we shall henceforth simply write:

subclass(?CLASS1, ?CLASS2).

From this notation we (but not the system, that does not analyze variable names) are immediately able to see that *subclass* is a predicate of arity 2, which takes some class as both of the arguments. This is easily understood with only a quick glance. When required for understanding, we will still provide the domain limits of arguments and additional information.

2) *Existing Knowledge*

In our implementation, we use an extended full Cyc ontology and KB, similar to that released as ResearchCyc, as the existing knowledge base. This is far too big (millions of assertions), to be defined in any detail here. For this reason, we define just the necessary concepts for the purposes of explanation of the approach. In this section we will not define everything needed, for the imaginary KB to be accurate as a partial model of the world and consistent, as we did in the previous section. We will be operating with the following classes:

User, User1, User2, Place, PublicPlace, Restaurant, Cuisine, FrenchCuisine, FoodOrDrink, Food, Drink, Animal, Meat, Bread, Vehicle, Car, Baguette, Visit, Restaurant1, Coffee (14)

And predicates:

probableUserLocation(?USER, ?PLACE),
userLocation(?USER, ?PLACE),
restaurantServesCuisine(?RESTAURANT, ?CUISINE),
menuItem(?RESTAURANT, ?FOODORDRINK),
cuisineIncludes(?CUISINE, ?FOODORDRINK),
userPlaceVisit(?USER, ?PLACE, ?NUMBER),
userOfVisit(?VISIT, ?USER), timeOfVisit(?VISIT, ?TIME)
placeOfVisit(?VISIT, ?PLACE)
orderedFood(?VISIT, ?FOODORDRINK),
assert(?FORMULA),
disjointWith(?CLASS, ?CLASS),
visitDuration(?VISIT, ?NUMBER) (15)

Where both users are instances of a class *User*:

$$is(User1, User) \wedge is(User2, User) \qquad (16)$$

Places and foods are connected into a hierarchical subclass structure:

$$subclass(PublicPlace, Place) \wedge subclass(Duck, Animal) \wedge$$
$$subclass(Restaurant, PublicPlace) \wedge$$
$$is(Restaurant1, Restaurant) \wedge subclass(Food, FoodOrDrink) \wedge$$
$$subclass(Drink, FoodOrDrink) \wedge subclass(Coffee, Drink) \wedge$$
$$subclass(Meat, Food) \wedge subclass(Bread, Food) \wedge$$
$$subclass(Baguette, Bread) \wedge is(FrenchCuisine, Cuisine) \wedge$$
$$menuItem(Restaurant1, Coffee) \wedge$$
$$cuisineIncludes(FrenchCuisine, Baguette)$$
$$disjointWith(Car, FoodOrDrink) \qquad (17)$$

There is an additional rule that can populate additional types of food by taking the subclasses of animals and creating a subclasses of food from them using a predicate *assert*, which has a special function to be able to assert its contents into the KB[7]:

$$\forall x\ (subclass(x, Animal) \Rightarrow assert('is(\$xMeat, Class)') \wedge$$
$$assert('subclass(\$xMeat, Meat)') \qquad (18)$$

Our pre-existing knowledge structure can be now used by the KA rules and meta-knowledge (sub-section 3) and is visually presented on Figure 3.

[7] This is not the notation used in the implemented system, but conveys its effect.

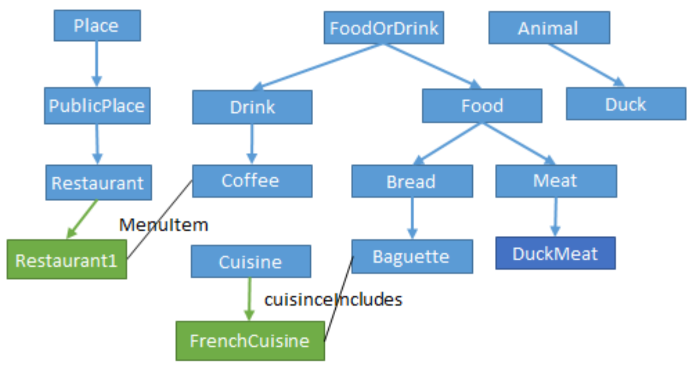


Figure 3: Visual representation of the example pre-existing knowledge

3) *KA Knowledge.*

In the previous two sections we defined the supporting concepts and, based on them, the pre-existing knowledge, which will suffice to support the KA process. For this, we need to define a few new meta-classes:

Formula – All logical assertions in the knowledge base are automatically instances of this class. For example*:*

$$is(\text{‘}is(Predicate, Class)\text{’}, Formula). \tag{19}$$

And predicates:

$$\begin{gathered} unknown\ (?FORMULA),\ known\ (?FORMULA),\\ list(?SOMETHING, ?SOMETHING),\\ ccWantsToAsk(?USER, ?FORMULA),\\ ccWantsToAskWithSugg(?USER, ?FORMULA,?LIST),\\ ccWantsToComment(?USER, ?FORMULA) \end{gathered} \tag{20}$$

Where *known* and *unknown* predicates are true when the logical formula given as their argument exists, or does not exist respectively in the manifest KB or in what can be readily inferred from it using backward-chaining inference. The logical function *list* can represent a list of concepts, through list of lists structures, for example: *list(concept1, list(concept2, concept3)*. And *ccWantsTo…* predicates which allow the system to remember its intention to produce a question or comment addressed to a specific user.

Now we define a set of KA rules, which can be written specifically to enable the production of questions for a narrow context, or which can be general KA rules like:

$$\forall c\ \forall P i\ \exists s\ \forall i2\ \forall u\ (is(i, c) \wedge P(i, s) \wedge is(i2, c) \wedge unknown(\text{“}\exists s2\ P\ (i2,s2)\text{”}) \Rightarrow CCWantsToAsk(u, \text{“}\$P(\$i2,\$\$x)\text{”})) \tag{21}$$

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 doesn't have any assertion using this predicate. If we take an example from our example KB and imagine we add Restaurant2, the premises of the rule can be satisfied like this:

$$(is(Restaurant1, Restaurant) \wedge menuItem(Restaurant1, Coffee) \wedge is(Restaurant2, Restaurant) \wedge unknown(\text{“}\exists s2\ menuItem\ (Restaurant2,s2)\text{”})$$

Which then produces the consequent: *ccWantsToAsk(User1, "menuItem(Restaurant2, $x)")*, which can be sent to the NL converter to ask the question, "What is on the menu in Restaurant2". The rule (21) effectively detects when there is an instance in the KB that doesn't 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 above 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 a great many questions, including, in some cases, many irrelevant ones. To mitigate this in our implementation we have additional rules that can suppress questions on some predicates or for whole parts of the KB. While we defined rule 21 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. To explain the examples from Table I, we are still missing few KA rules:

$$\forall u\ \forall p(probableUserLocation(u, p) \Rightarrow CCWantsToAsk(u, \text{“}userLocation(\$u, \$p)\text{”})) \tag{22}$$

$$\forall u\ \forall p(userLocation(u, p) \wedge is\ (p, Restaurant) \Rightarrow CCWantsToAsk(u, \text{“}restaurantServesCuisine\ (\$p,\$\$x)\text{”})) \tag{23}$$

$$\forall u\ \forall p\ \forall c\ \forall f\ (userLocation(u, p) \wedge restaurantServesCuisine\ (p, c) \wedge cuisineIncludes(c, f) \Rightarrow CCWantsToAsk(u, \text{“}menuItem\ (\$p, \$f)\text{”})) \tag{24}$$

$$\forall u\ \forall p\ \forall t\ (userPlaceVisit(u, p, t) \Rightarrow assert(\text{“}is\ (\$u\$p\$t, Visit)\text{”}) \wedge assert(\text{“}userOfVisit(\$u\$p\$t, \$u)) \wedge assert(\text{“}timeOfVisit(\$u\$p\$t, \$t)) \wedge assert(\text{“}placeOfVisit(\$u\$p\$t, \$t))) \tag{25}$$

$$\forall v\ \forall u\ (is(v, Visit) \wedge userOfVisit(v, u) \wedge unknown(\text{“}orderedFood\ (\$v, \$\$x)\text{”}) \Rightarrow ccWantsToAsk(u, \text{“}orderedFood(\$v, \$\$x)\text{”}) \tag{26}$$

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 is is simply controlled by the number and content of the rule premises.

C. *Context*

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 questions is of crucial importance. For this reason, much of our KB content is actually user context, which can be used with the KA rules as the rest of the knowledge. One example of contextual knowledge, seen above along with itsinfluence on the KA process, is the *userLocation(?USER,*

?LOCATION) predicate. Besides *userLocation*, there is much additional external information that is worth bringing in (see definition 15). Moreover, the system also uses internal context, which is more based on its knowledge about the user.

1) External context

It makes sense for the KA system to understand as much about the context of the user as possible. The central and most important piece of contextual knowledge in our approach is the user's location and the duration of stay at this location. For this function, we use an improved implementation of the stay-point detection [20] algorithm, which is able to cluster raw GPS coordinates and detect when users are moving, or stay at a particular location for a while (Figure 4). A paper explaining the customizations and improvements of SPD used for CC and usable for other location-centric applications is in preparation.

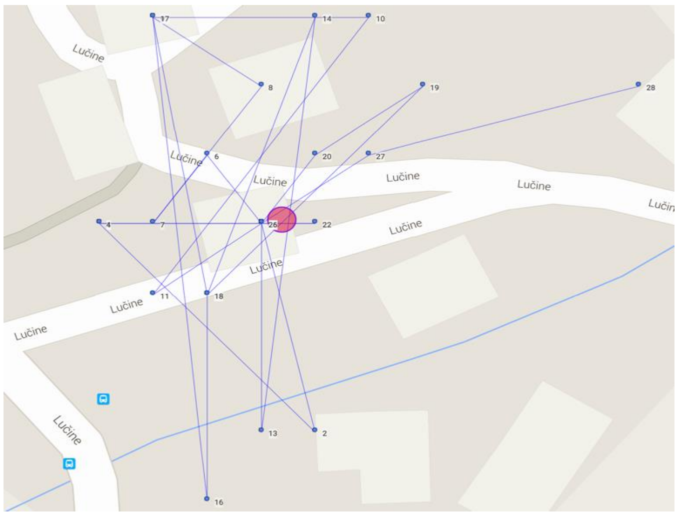


Figure 4: Clustering of raw GPS data into a location, to provide stayAtLocation context

Once the raw GPS coordinates are identified as belonging to a location p', the functional component (orange in Figure 2), checks the location provider (Foursquare in this case) API to determine a location identifier p for p' and asserts:

probableUserLocation(u, p)

And once user confirms the location, the functional component keeps asserting the time the user has stayed at that location, so the questions can vary based on stay duration:

visitDurationAt(v, l)

Where v is a reified visit event. This allows the system to ask something at arrival, and something else, which only makes sense then, later. Questions about the quality of food that someone ordered in a restaurant is an example of that latter case. Besides the location, the system uses other external context:

userTimezone(u,t),
userLocalTime(u, lt),
userPartOfDay(u,d)[Dusk, Dawn, Morning, BeforeNoon, MidDay, Evening, Night],
userActivity[Sitting, Walking, Running, Cycling, InVehicle],
placesInVicinity(?LIST)

This knowledge is pretty easily gathered from the phone, which provides the main UI for the system and allows us to trigger the right questions and information at the right time.

2) Internal Context

Besides the external context, defined above, we also use internal context, which is simply a specific set of KA rules and KB knowledge directly relevant to the users themselves. These KA methods for user-specific knowledge vastly improve the relevance of the questions presented. This generated question includes those about languages spoken, profession, interests, food liked, etc.:

userAge(?USER, ?TIME),
userSpeaksLanguage(?USER, ?LANGUAGE),
userInterest(?USER, ?ACTIVITY)
…

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

D. NL to Logic and Logic to NL conversion

We have discussed how the questions in logical form are generated using an inference engine and KA rules. These formulas are understandable by knowledge engineering and math experts, but are not at all appropriate for 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 into natural language and (at least elements) from natural language to logical form, when users answer.

In addition to the knowledge presented in the previous sections, the KB required for our method includes NL knowledge. Because NL generation and conversion is not the main focus of this paper, we present here only simplified version which explains the basic concepts involved. The actual Curious Cat implementation is based on CYC NL [21] and SCG [22] and consists of more than 90 additional assertions and rules beyond those in the baseline Cyc system to handle language generation.

1) Logic to NL

Each of the concepts in the KB, can be named using a standard String:

nameString(?SOMETHING, ?STRING),
nameString(restaurant1, "L'Ardoise") (27)

For example: *nameString(user1,"Luka Bradeško").* For more complicated language structures, we can encode words as concepts inside the knowledge base[8]:

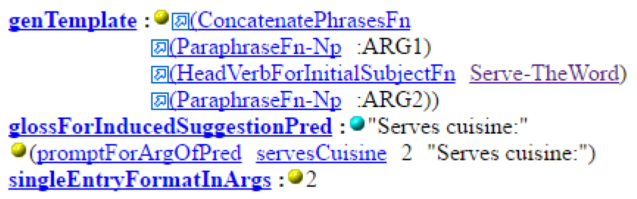
is(Cuisine-word, Word), is(French-word, Word),
is(Serve-word, Word),
singularNotation(Serve-word, "serve"),
singularNotation(Cuisine-word, "cuisine"),
pluralNotation(Cuisine-word, "cuisines"),
denotation2(FrenchCuisine, list(French-word, Cuisine-word)),
denotation2(FrenchCuisine, list(French-word, "food")),

[8] Note that "French-word" here means the word, "French", not the class of words in French.

denotation1(FrenchCuisine, French-word)
denotation2(restaurantServesCuisine, list("$arg1",
Serve-word, "$arg2")) (28)

With the assertions above, the system can already convert the logical formula: *restaurantServesCuisine(restaurant1, FrenchCuisine)* into the appropriate English form: "L'Ardoise serves French cuisine", or: "L'Ardoise serves French food".

Now, to generate the following question: *restaurantServesCuisine(restaurant1, ?X)* there are two ways to do it. The first one, declarative, works in the same fashion as the same as the logic without variables and can be produced with our KB defined in 28: "L'Ardoise serves ________.". With additional knowledge it is also possible to generate in the interrogative mode: "What kind of cuisine does L'Ardoise serve?". For this, we need a much more detailed linguistic KB than defined above[9] In our Cyc-based implementation, *Serve-word* alone consists of some 55 language generation connected assertions (not including the actual uses of the word). And the predicate *restaurantServesCuisine* has 57 assertions on its own.



```
genTemplate : (ConcatenatePhrasesFn
                (ParaphraseFn-Np :ARG1)
                (HeadVerbForInitialSubjectFn Serve-TheWord)
                (ParaphraseFn-Np :ARG2))
glossForInducedSuggestionPred : "Serves cuisine:"
(promptForArgOfPred servesCuisine 2 "Serves cuisine:")
singleEntryFormatInArgs : 2
```

Figure 5: Actual NL generation assertion example for the corresponding predicate for *restaurantServesCuisine* from our implementation (Cyc)

There is an additional, perhaps unobvious, benefit from explicit NL knowledge. When we are missing some knowledge on how to convert some predicate or concept to English, the system can simply ask users, how to say something in the language that particular user knows, and thus the system can make its own NL generation knowledge part of the KA process as well.

2) *NL to Logic*

While simple processing based on the same NL knowledge suffices to interpret isolated terms and denotational phrases from user responses, vonverting general NL expressions back to logic is much trickier than the other way around, because natural language is much more ambiguous. The process exceeds the scope of the paper and is described in more detail in [22]. A ful language-to-logic component was only partly deployed on a trial basis within the tested system.

Consider inverting the example defined in 28: The system is presented with the statement "L'Ardoise serves French food". Because all the textual representations of the concepts are indexed, it is able to find the concepts:

- restaurant1
- restaurantServesCuisine
- FrenchCuisine

[9] An astute reader will have noticed that the sketched KB content does not specify how to produce the third person present for Serve-Word used in the example.

Because of the *is* and *subclass* knowledge of *restaurant1* and *FrenchCuisine* (definition 17), and the *argIs* and *argClass* domain limits of *restaurantServesCuisine* (def. 15), the inference engine is actually able to construct: *restaurantServesCuisine(Restaurant1, FrenchCuisine).*

Now consider, that we have language representation knowledge for the predicate *menuItem* which is the same as for *restaurantServesCuisine*:

is(Coffee-word, Word)
singularNotation(Coffee-word, "coffee"),
denotation1(Coffee, Coffee-word)),
denotation2(menuItem, list("$arg1",
Serve-word, "$arg2")) (29)

Then, for the same statement, our system will find these possibilities:

- *Restaurant1*
- *restaurantServesCuisine, menuItem*
- *FrenchCuisine*

At this moment it needs to check all the constraints again, and filters out *menuItem* because it needs a subclass of *FoodOrDrink*, which is not in the list of found concepts. This logical statement would be invalid due to argument constraints in our KB: ~~*menuItem(Restaurant1, FrenchCuisine)*~~. Only the predicate *restaurantServesCuisine* can actually replace its arguments with our available concepts and still be valid in our KB.

These two examples are the simplest possible kinds of sentential NL conversion that could happen. When there are many more possible concepts which need to be combined, the complexity of the problem quickly explodes, or stays ambiguous even given constraints. For this reason, our actual Cyc SCG implementation contains much more complex KB structures and as well patterns which help with the conversion and a high-speed parser, in Java, that performs the needed search.

E. *Consistency Check and KB placement*

In the examples and definitions in the previous sections, we saw how we can employ the inference engine to deduce various facts, using forward-chaining inference and then automatically insert the deduced knowledge into the KB. Its knowledge 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 then inference tries to prove or satisfy. For example:

class (?X, Food) (Q1)

Will return the following if queried over our example KB:

Bread, Meat, Baguette, DuckMeat (R1)

Or the (somewhat odd but valid) query:

is(Restaurant1, Place) $\wedge$ *class(Restaurant, PublicPlace)* (Q2)

Will return *TRUE.*

Once CC finds a question to ask, for example "What kind of cuisine does Restaurant2 serve?":

restaurantServesCuisine (Restaurant2, ?CUISINE) (30)

With the argument constraints on the predicates:

argIs(restaurantServesCuisine, 1 Restaurant) ∧
argIs(restaurantServesCuisine, 2 Cuisine) (31)

And then let's say, the user answers with "French Cuisine", as in our example conversation (Table I), Then we can ask the inference engine the following type of general query:

denotation(?TERM, $answer) ∧
argIs($pred, $argPos, ?ARGIS) ∧
is(?TERM, ?ARGIS) ∧ [
(argClass($pred, $argPos, ?ARGCLASS) ∧
subclass(?TERM, ?ARGCLASS)) ∨
(unknown(∃?ARGCLASS(argClass($pred, $argPos, ?ARGCLASS)*)))]* (Q3)

Where we replace the meta-variables *$answer* ("French cuisine), *$pred* (*restaurantServesCuisine*), *$argPos* (the position of variable in the query - 2), with our values from the question and user's answer, so we get the following actual query:

denotation(?TERM, "French cuisine") ∧
argIs(restaurantServesCuisine, 2, ?ARGIS) ∧
is(?TERM, ?ARGIS) ∧ [
(argClass(restaurantServesCuisine, 2, ?ARGCLASS) ∧
subclass(?TERM, ?ARGCLASS)) ∨
(unknown(∃?ARGCLASS(argClass(*restaurantServesCuisine*, 2, ?ARGCLASS)*)))]* (Q4)

If this query is asked in our small KB, we get the following results:

?TERM,	*?ARGIS,*	*?ARGCLASS*
FrenchCuisine,	*Cuisine,*	*/*

(R2)

Because we got results from this query we can immediately know that the answer is consistent with the KB and it's safe to assert it into the knowledge base as: *restaurantServesCuisine(Restaurant2, FrenchCuisine).* Here it is worth noting that we allow the variable ?ARCLASS to be unbound because of the logical disjunction in the query. In the real implementation this is not allowed, so we add additional disjunction to bind it to the concept *Nothing*.

At this point we have seen an example of how to add new knowledge when the concept that a user answered with exists in the KB and the answer is actually structurally valid. But what would happen if user should say "Italian cuisine", which we don't have in the KB. In this case the query above would return nothing. This would happen as well, if the user should say "car". So, when the validation query doesn't return results (as would happen for "car"), we need to separately check whether the concept exists:

denotation(?TERM, "car") (Q5)

If it does (i.e. the query 5 above returns the resulting concepts), then the answer is actually invalid on structural grounds (the term it includes can't be used a viable answer). But, if the concept doesn't exist yet (nothing returned, as would be the case for "Italian cuisine"), then we can simply create it, together with its NL denotation and assert it also using our question predicate.

1) Detailed placement in the KB

Above we saw how new knowledge is added to the KB. The location in the "KB graph" is determined by the *argIs* and *argClass* assertions on the question predicate. Consider another example question: *menuItem(Restauratn1, ?X).* Because of the *argClass(menuItem,2 FoodOrDrink)*, the answer must be a subclass of FoodOrDrink concept. Let's say, our answer on that question is: "Strange coffee".

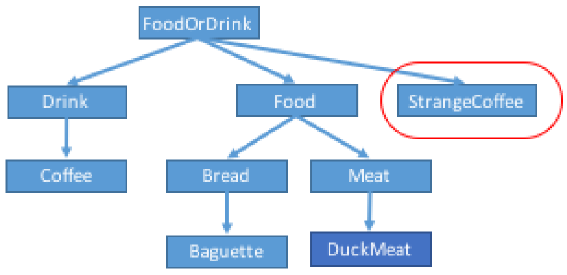


Figure 6: FoodOrDrink part of the class hierarchy from our example KB

Now the system goes through the steps described in the section E (Answer Validation and KB Placement), and, because of *argClass* assertions, enters *StrangeCoffee* concept as a subclass of FoodOrDrink. From the steps described up to now, the KA process only knows that the missing part must be FoodOrDrink, and nothing else (Figure 6). While this is logically valid, it is not detailed enough to satisfy our KA requirements. For this reason, when the concept doesn't already exist, or is not detailed enough (too high in the class hierarchy), we can issue the additional query:

subclass(?TERM, FoodOrDrink) (Q6)

Which, for our example knowledge base returns:

?TERM: FoodOrDrink, Drink, Coffee, Food, Bread, Baguette, Meat, DuckMeat (R3)

Now, we have various options. A) Ask the user, which one of these StrangeCoffee is (excluding the main class FoodOrDrink). For example: "What describes it in most detail?", or: "Is Strange coffee a drink, coffee, bread, baguette, meat or duck meat". B) Ask for the first level subclasses first, then for the next level, etc. until the user doesn't select any of the options: 1. "Is Strange coffee a type of drink or food?", 2. "Is Strange Coffee a type of coffee?". Or C) (which is usually the best option) We can scan all of the resulting concepts from R3, for their NL *denotations* and then match the strings to "Strange Coffee", to see which one fits the best. In this case, only the *Coffee* concept provides a partial match. So we can immediately ask: "Am I right that Strange Coffee is a type of coffee?". If the user agrees, we can, in addition to *(subclass, StrangeCoffee, FoodOrDrink)*, add: *(subclass, StrangeCoffee, Coffee)*. This results in our new concept being located in the KB at the most descriptive place (Figure 7).

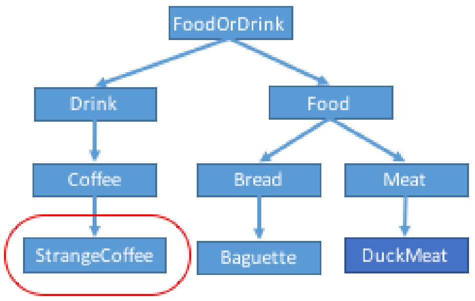


Figure 7: New position of the StrangeCoffee concept in our KB

F. *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. But, if we want to lower the cost of KA, or increase the speed of acquisition, which is the main purpose of our approach, we need access to a crowd. This brings in a new set of problems, such as:

- User privacy
- Users making deliberately false claims or having mistaken ideas about the world, and
- The ever-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 8). In the Cyc system, these contextual KB structures are called Microtheories [23].

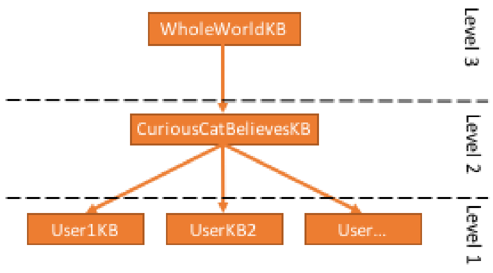


Figure 8: Hierarchical structure of Knowledge Bases

The sub-KBs which are higher up in the hierarchy cannot see the knowledge that is lower down. But the KBs which are on lower levels, contain all the knowledge of their ancestors from the higher levels. For example, let's say *WholeWorldKB* has the knowledge defined in 1, *CuriousCatBelievesKB* has knowledge defined in 2, *User1KB* has knowledge defined in 3 and *user2KB* has knowledge defined in 4. Following our KB structure from Figure 8, we can see that:

- *WholeWorldKB* contains assertions from definition1
- *CuriousCatBelievesKB* contains assertions from definitions 1 and 2.
- *User1KB* contains assertions from definitions 1, 2 and 3
- *User2KB* contains assertions from definitions 1, 2 and 4

Following this structure, it becomes obvious, 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 other's 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 won't be affected. This way, user's private information is also protected.

Now the question is: how can users then see other's users answers and thus how can the system benefit from crowdsourcing. 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 hierarchy. We have two approaches for that: 1) Crowdsourcing through repetition, and 2) Crowdsourcing through voting.

1) *Crowdsourcing through repetition*

This approach (the crowdsourcing component from Figure 3), counts the number of identical assertions in all the sub-KB's on the same level. Once the specific assertion count goes above some threshold, we can simply move the knowledge up (via "lifting rules" into the *CuriousCatBelievesKB*), and thus make it visible for all the users. After the knowledge is in the public KB, crowd users can start voting on it (see Crowdsourcing through voting subsection.)

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 E (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 Restaurant1 has them on the menu. The contextual KB User1KB would then get this assertion *menuItem(Restaurant1, 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 the higher level.

On the other hand, if User1 answers *menuItem(Restaurant2, Coffee)*, 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. For newcomers to Restaurant1, the system will already know that they have coffee on the menu.

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 user's 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: “Is it true that Restaurant1 has coffee 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 as well for previously promoted knowledge described in the previous subsection (Crowdsourcing through repetition) and for detecting when the world changes and something that was true in the past is not true anymore.

IV. Validation

During the course of 3.5 years the Curious Cat project was online, we had a total of 600 registered users, who checked-into 5,536 locations and answered 55,992 questions, of which 8,492 were voting questions (7,456 positive and 1,036 negative votes) and **47,500** real pieces of knowledge added to the KB. These additions triggered additional 338,068 assertions to be added through forward-chained inference, so altogether we gathered **394,060** new pieces of knowledge, which can be separated into facts (238,437 assertions) and additional derived question generating rules and questions (9,631). Results are presented hierarchically in Table III.

There are two observations from these assertion counts (Table III) that we need to explain:

A. Number of Question generating rules and assertions

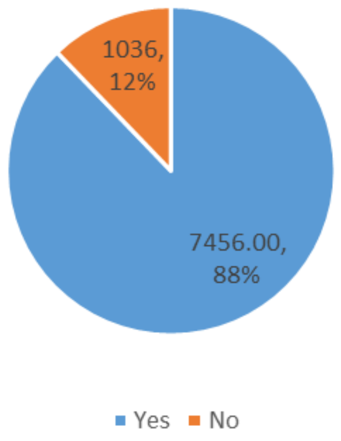
B. Number of Yes/No votes

A) From the results, we can see the ratio of answers/new questions is 47,500/9,631. This could easily lead to the wrong conclusion: that new questions are not generated as quickly as the answers are coming in and thus that the system will become saturated very soon and not gather any new knowledge. This is an incorrect conclusion for two reasons. 1. Many of the KA rules are written in such a way, that when a question has been answered, the actual question is retracted. So we are now see only the number of as-yet-unanswered questions, which were generated by the system. 2. Some of the existing and also some of the newly generated rules, are used by backward inference, which only executes at query time, when the system checks what it wants to ask next. Because the results of these rules are not asserted into KB beforehand, and due to their complexity and number, we don’t have a straight forward way to check how many unanswered question the system currently has at hand.

TABLE III HIERARCJICALLY PRESEMTED NUMBER OF ACQUIRED KNOWLEDGE ITEMS

All newly acquired knowledge				
394,060				
Directly asserted by users			*Acquired by Inference*	
55,992			338,068	
Voting answers		**Real Answers**	*Q rules*	*Facts*
8,492		47,500	9,631	328,437
Approvals	*Rejections*			
7,456	1,036			

B) The Yes/No votes are not voting whether the statement is consistent or not, but whether it’s true or not. Because Curious Cat always rejects answers that are manifestly inconsistent with other content in its KB, it is unlikely that it would contain inconsistent knowledge. But claims can be still be untrue, even if thy’re 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 other users provide, are mostly true. This could be taken as a hint towards the precision of the truthfulness of the system, but we need to take into the consideration that more users can vote for the same assertion, so the effective precision measured this way would be higher. The voting mechanism thus hides from the public knowledge base 37 of the assertions where users were unable to agree on truth and 593 assertions which were voted as untrue by majority.



Graph 1: Ratio of true versus not true statements as voted on by users

Here are a few examples of the removed assertions:

- Salad is a type of vegetable *[subclass(Salad, Vegetable-Food)]*
- Mass of someone is similar to mass of a male [typesSimilarWRTTypeViaBinRel(Person, Male, Mass, massOfObject)]

As a sanity check of the KB, we randomly picked 100 newly acquired assertions, and assessed them whether theywere:

- 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 IV below:

TABLE IV RESULTS OF MANUAL EVALUATION ON 100 RANDOMLY PICKED ASSERTIONS

Valid	***True***	***Useful***
100	96	95

The results of this counting are not surprising, since the system doesn't 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:

1. One error was because the price of the coffee changed since the last check with users
2. Once, there were two concepts with exactly the same name and were both subclasses of *Organization*, so the users and also the system didn't manage to distinguish them and picked the wrong one. For users everything seemed perfectly correct even though a logical error resulted, and
3. Twice the user entered a complex sentence instead of a name of the concept and then forced the system to create a concept with that name.

Regarding the usefulness of retrieved knowledge, there were two answers, which related to the internal KB mechanism to handle events and were thus not really useful for the end user. The rest were mostly not really useful assertions from the users, such as the name of the spider they have in the corner of the room.

V. CONCLUSION

In the paper we presented Curious Cat, a novel Conversational Agent and KA and system, which is able to gather new knowledge of a very high quality in a never ending fashion [4]. This is achieved using an existing knowledge base, user current and past context and finally by using targeted crowdsourcing methodology in a natural language. The CC method of using highly focused context and thus targeting the right users at the right time, had not to our knowledge been tried before and allowed the collection of knowledge which would otherwise be inaccessible. The proposed system was online as an experiment for 3.5 years. During this time, it collected a substantial amount (394,060 assertions) of consistent and highly accurate knowledge and thus proved the approach to be feasible and worth exploring. The implementation of the prototype required approximately 2.5 person years, on top of ~930 person years spent on Cyc system. While we implemented our prototype on top of Cyc [1], the approach itself is general and can be applied to any KB or/and inference engine.

From the validation (Section IV), we can see that the resulting knowledge has high quality and is easily and inexpensively gathered from non-expert users, while they are having a conversation about some secondary task (helping the users). The validation examples also hinted at potential problems, which are to be addressed in future work.

For problem 1, our plan is to make use of the time constraints and assertion meta-knowledge in the Cyc KB and then have KA rules, which ensure that after some knowledge has not been confirmed for a while, that the system checks with users in context about the claim's validity. Problem 2 is already being tackled by some extent, since when there are more possible answers with the same NL representation, Curious Cat asks the user, which one of the options he meant, using alternative NL presentations when they are available in the system. There is a border scenario, where more concepts have exactly the same NL name. In this cases we plan to add an additional description, in the follow up question, where Curious Cat will try to describe each concept with some unique predicate that holds for each. The identified problem number 3, we plan to tackle simultaneously while improving the conversational client rules. One of the promising directions for the future work is, for Curious Cat would become a conversational engine, which would make it easy to construct knowledgeable chatbots. This is quite similar to 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 users. As part of this approach, the answers of the users, which are not found in the KB, will not simply be asserted as the proper concepts, but first checked to determine whether they can be parsed into a more complex logical sentence and thus used as part of the conversation. KA and Conversation rules can thus be enabled to trigger on these more complex statements as well.

ACKNOWLEDGMENT

We thank to all of our 600 users who contributed to the experiment and the KA system, and Dr Dave Schneider in particular and Cycorp management and members of the Cycorp technical staff (including Chris Deaton and Dr David Baxter) more generally for licensing, and for kind assistance with APIs, KR advice and other assistance as we constructed and tested Curious Cat. We also thank Abe Hsuan and Dane Sulic for advice and design assistance respectively.

Note: The overwhelming majority of this work, except writing, was done while Witbrock was acting as a principal at Curious Cat Company and concurrent with his time at Cycorp Inc.

The authors acknowledge the European Commission for its support and partial funding and the partners of the research project: H2020-636160 Optimum.

REFERENCES

[1] D. B. Lenat, "Cyc: A Large-Scale Investment in Knowledge Infrastructure," *Communications of the ACM,* vol. 38, no. 22, 1995.

[2] C. Matuszek, J. Cabral, M. WItbrock and J. DeOliviera, "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.*

[3] L.-J. Zang, C. Cao, Y.-N. Cao, Y.-M. Wu and C.-G. Cao, "A Survey of Commonsense Knowledge Acquisition," *Journal of Compter Science*

and Technology, vol. 28, no. 4, pp. 689-719, July 2013.

[4] T. Mitchell, W. Cohen, E. Hruschka, P. Talukdar, J. Betteridge, A. Carlson, B. Dalvi, M. Gardner, B. Kisiel, J. Krishnamurthy, N. Lao, K. Kazaitis, T. Mohamed, N. Nakashole, E. Platanios, A. Ritter, M. Samadi, B. Settles, R. Wang, D. Wijaya, A. Gupta, X. Chen, A. Saparov, M. Greaves and J. Welling, "Never-Ending Learning," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI-15)*, 2015.

[5] M. WItbrock, D. Baxter, J. Curtis, D. Schneider, R. Kahlert, P. Miraglia, P. Wagner, K. Panton, G. Matthews and A. Vizedom, "An Interactive Dialogue System for Knowledge Acquisition in Cyc," in *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, 2003.

[6] K. Forbus, C. Riesback, L. Birnbaum, K. Livingston, A. Sharma and L. Ureel, "Integrating natural language, knowledge representation and reasoning and analogical processing to learn by reading," in *Proc. the 22nd National Conf. Artificial Intelligence*, 2007.

[7] A. Sharma and K. Forbus, "Graph based reasoning and reinforcement learning for improving Q/A performance in large knowledge-based systems.," in *Proc. the AAAI Fall Symposium Series*, 2010.

[8] P. Singh, T. Lin, E. Mueller, G. Lim, T. Perkins and W. Zhu, "Open Mind Common Sense: Knowledge acquisition from the general public," in *Cooperative Information Systems Oct. 30-Nov. 1 2002*, 2002.

[9] Y. Kuo and J. Hsu, "Goal-oriented knowledge collection," in *The AAAI Fall Symposiym Series, Nov. 2010*, 2010.

[10] S. D.S. Pedro and H. J. Estevam R., "Collective intelligence as a source for machine learning self-supervision," in *Proceedings of the 4th International Workshop on Web Intelligence & Communities. In conjunction with WWW2012*, 2012.

[11] S. Pedro, A. Appel and E. Hruschka, "Autonomously reviewing and validating the knowledge base of a never-ending learning system," 2013.

[12] R. Speer, J. Krishnamurthy, C. Havasi, D. Smith, H. Lieberman and A. Keneth, "An interface for targeted collection of common sense," in *Proceedings of the 14th International Conference on Intelligent User Interfaces*, 2009.

[13] E. Hruschka and S. D. S. Pedro, "Conversing Learning: Active Learning and Active Social Interaction for Human Supervision in Never-Ending Learning Systems," in *Advances in Artificial Intelligence -- IBERAMIA 2012: 13th Ibero-American Conference on AI, Cartagena de Indias, Colombia, November 13-16, 2012*, Berlin, Heidelberg, 2012.

[14] R. Speer, "Open mind commons: An inquisitive approach to learning common sense," in *Workshop on Common Sense and Intelligent User ...*, 2007.

[15] R. Speer, H. Lieberman and C. Havasi, "AnalogySpace : Reducing the Dimensionality of Common Sense Knowledge," *AAAI,* vol. 23, pp. 548-553, 2008.

[16] R. Wallace, "The elements of AIML style," Alice AI Foundation, 2003.

[17] B. Wilcox, "Beyond Façade: Pattern Matching for Natural Language Applications," 2011.

[18] K. Coursey, "Living in cyn: mating aiml and cyc together with program n.," 2004.

[19] A. Saenz, "Cleverbot Chat Engine Is Learning From The Internet To Talk Like A Human," 13 Jan 2010. [Online]. Available: http://singularityhub.com/2010/01/13/cleverbot-chat-engine-is-learning-from-the-internet-to-talk-like-a-human/. [Accessed 2016].

[20] L. Quannan, Y. Zheng, X. Xie, Y. Chen, W. Liu and W.-Y. Ma, "Mining User Similarity Based on Location History".

[21] D. S. B. S. N. G. B. &. S. D. Baxter, "Interactive Natural Language Explanations of Cyc Inferences," in *AAAI 2005: International Symposium on Explanation-aware Computing*, 2005.

[22] D. W. M. Schneider, "Semantic Construction Grammar : Bridging the NL / Logic Divide," in *WWW '15 Companion Proceedings of the 24th International Conference on World Wide Web*, FLorence, 2015.

[23] Cycorp, "Foundations of knowledge representation in cyc – microtheories.," Cyc 101 Tutorial. Cycorp Corporation, 2012.