CASAM

FP7-217061

Computer-Aided Semantic Annotation of Multimedia

Deliverable D3.4

Meta-level reasoning engine, Report on meta-level reasoning for disambiguation and preference elicitation

---

Editor(s):	Oliver Gries, Ralf Möller, Anahita Nafissi, Maurice Rosenfeld, Kamil Sokolski, Michael Wessel
Responsible Partner:	TUHH
Status-Version:	Final Version
Date:	26/10/10
EC Distribution:	

Project Number:	FP7-217061
Project Title:	CASAM

Title of Deliverable:	Meta-level reasoning engine, Report on meta-level reasoning for disambiguation and preference elicitation
Date of Delivery to the EC:	

Workpackage responsible for the Deliverable:	WP3 Knowledge representation and reasoning for multimedia interpretation
Editor(s):	Oliver Gries, Ralf Möller, Anahita Nafissi, Maurice Rosenfeld, Kamil Sokolski, Michael Wessel
Contributor(s):	Oliver Gries, Ralf Möller, Anahita Nafissi, Maurice Rosenfeld, Kamil Sokolski, Michael Wessel
Reviewer(s):	Chris Bowers (University of Birmingham), Sergios Petridis (NCSR "Demokritos")
Approved by:	All partners

Abstract:	A mechanism for meta-level interpretations is presented with the aim to disambiguate interpretation alternatives. This is done by generating queries out of the interpretation alternatives and stating them to the user or other agents. Queries are ranked by an importance value representing the value of the answer. The query generation mechanism is explained followed by the processing of the response and a detailed description of query types.
Keyword List:	Meta-Reasoning, Multimedia Interpretation, HCI, Multi-Agent Communication

Document Description

Document Revision History

Version	*Date*	*Modifications Introduced*	
		Modification Reason	*Modified by*
V0.1	30/09/10	Version for internal review	Oliver Gries, Ralf Möller, Anahita Nafissi, Maurice Rosenfeld, Kamil Sokolski, Michael Wessel
V0.2	18/10/10	Final Version	Oliver Gries, Ralf Möller, Anahita Nafissi, Maurice Rosenfeld, Kamil Sokolski, Michael Wessel
V0.3	26/10/10	The annex was added	Ralf Möller, Maurice Rosenfeld

Executive Summary

In [1] and [2] an agent was presented that builds interpretations upon multimedia annotations by incrementally consuming analysis results as well as input from a human annotator. These interpretations are based on background knowledge of a specific domain, e.g. an environmental domain as it was exemplarily chosen for the CASAM project. As a result of the interpretation process, multiple interpretation alternatives are possible. A preference measure for the alternatives is realised by a probabilistic scoring function. As an extension of the interpretation agent, a mechanism for meta-level reasoning is presented with the aim to disambiguate interpretation alternatives. This is achieved by generating queries from a set of interpretation alternatives and stating them to the human annotator. Queries themselves are ranked by an importance value, representing the benefit of an answer to a query for the disambiguation process.

After a revision of the interpretation process the query generation mechanism is explained, followed by a detailed description of different query types, together with the format they are communicated in. Furthermore, the processing of responses to queries is addressed.

Contents

1 Introduction

To speed up manual multimedia annotation processes, the CASAM project investigates the collaboration of human annotators and machine intelligence. In [1] a Reasoning-based Media Interpretation module (RMI) was defined as an agent that computes interpretations given a set of observations. Generated interpretations are ranked by a probabilistic scoring function, and the Abox with the highest score is communicated to the Human Computer Interface module (HCI) for displaying the results. The set of all possible interpretations is managed by a so-called *agenda*. During the ongoing annotation process, more and more interpretation Aboxes have to be managed by the agenda. If additional observations are received they have to be added to all Aboxes. Due to possible new interpretations the probabilistic score may change. A considerable amount of computational resources are required to manage this kind of branching. Therefore, the agent has an interest to disambiguate interpretation alternatives and delete them from its agenda, especially when interpretations have similar scores.

In order to achieve the disambiguation, the agent computes a so-called Abox difference between the interpretation Aboxes. The result of this difference operation is a set of Abox assertions, where each assertion is unique to a corresponding Abox. These difference assertions are asked to an annotator in the form of queries. There is, however, a big gap between the interpretations maintained by the agent and the queries that are presented to the user. While interpretations can denote complex relational structures, queries on the can only be presented to the user in a flat natural language form. A Screenshot of the natural language query produced by the CASAM system is shown in Figure 1.

Figure 1: CASAM prototype displaying query choices

To make the translation from Abox assertions to flat natural language questions, HCI needs to classify certain type of queries. In this document all the foreseen types are described and specific examples are given. From the RMI agent's point of view, query answers have a value for the automatic annotation process, but from HCI's point of view, queries are a burden as they distract the user in focusing on the manual annotation. Distractions can be expressed by abstract costs. Therefore, a utility function that represents the value of an answer to a particular query is needed. The value of information or degree of importance for obtaining an answer is communicated by a so-called *importance value*.

This enables HCI to balance the benefit of a query versus its costs.
An answer to a query is communicated from HCI to RMI in the form of assertions. In this deliverable it is also shown how the user responses affect the probabilistic score of Aboxes on the agent's agenda.
The challenges being investigated in this deliverable are the following:

1. Generation of queries, including the transformation of relational structures of explanations into "flat" representations.

2. Computation of importance values for queries.

3. Processing of answers to queries.

Already in the 1980s, research has shown that meta-reasoning is useful for controlling the reasoning process [3]. The need of self-adaptation arises when an agent operates in a dynamic environment, such as a collaboration with a user. The meaning of the term *meta-reasoning* is reasoning about reasoning. This is different from performing object-level reasoning, which refers in some way to entities external to the system. A system capable of meta-reasoning may be able to reflect, or introspect, i.e. to shift from meta-reasoning to object-level reasoning and vice versa [4].

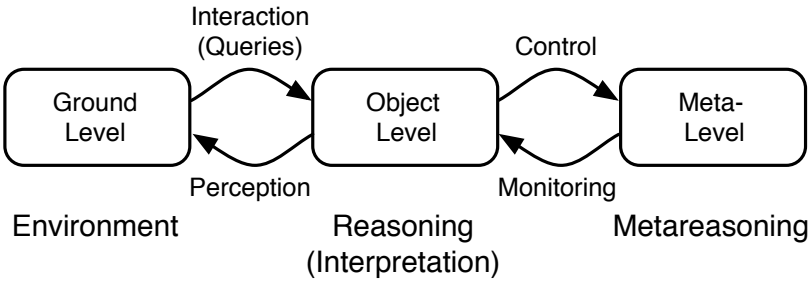

Figure 2: Schema of meta-reasoning

Figure 2 shows a general schema of meta-reasoning. In this deliverable it is shown that by ranking the interpretations, managing the interpretation agenda, and generation of queries, a *monitoring* and *control* functionality is achieved.

2 Preliminaries

In this section most important preliminaries already explained in detail in [5] are repeated in order to make this document self-contained.

2.1 Preliminaries on Description Logic

One of the main targets of the CASAM project is to support human annotators during their work in producing elaborate symbolic descriptions for video shots. Annotations are used for later information retrieval and require a representation language. We assume that a less expressive description logic (DL) should be applied to facilitate fast computations. We decided to represent the domain knowledge with the DL $\mathcal{ALH}_f{}^-(\mathcal{D})$ (restricted attributive

concept language with role hierarchies, functional roles and concrete domains). For details see [6].

In logic-based approaches, atomic representation units have to be specified. The atomic representation units are fixed using a so-called signature. A DL *signature* is a tuple $\mathcal{S} = (\mathbf{CN}, \mathbf{RN}, \mathbf{IN})$, where $\mathbf{CN} = \{A_1, ..., A_n\}$ is the set of concept names (denoting sets of domain objects) and $\mathbf{RN} = \{R_1, ..., R_m\}$ is the set of role names (denoting relations between domain objects). The signature also contains a component **IN** indicating a set of individuals (names for domain objects).

In order to relate concept names and role names to each other (terminological knowledge) and to talk about specific individuals (assertional knowledge), a knowledge base has to be specified. An $\mathcal{ALH}_f{}^-$ *knowledge base* $\Sigma_\mathcal{S} = (\mathcal{T}, \mathcal{A})$, defined with respect to a signature $\mathcal{S}$, is comprised of a terminological component $\mathcal{T}$ (called *Tbox*) and an assertional component $\mathcal{A}$ (called *Abox*). In the following we just write Σ if the signature is clear from context. A Tbox is a set of so-called *axioms*, which are restricted to the following form in $\mathcal{ALH}_f{}^-$:

(I)	Subsumption	$A_1 \sqsubseteq A_2$, $R_1 \sqsubseteq R_2$
(II)	Disjointness	$A_1 \sqsubseteq \neg A_2$
(III)	Domain and range restrictions for roles	$\exists R.\top \sqsubseteq A$, $\top \sqsubseteq \forall R.A$
(IV)	Functional restriction on roles	$\top \sqsubseteq (\leq 1\, R)$
(V)	Local range restrictions for roles	$A_1 \sqsubseteq \forall R.A_2$
(VI)	Definitions with value restrictions	$A \equiv A_0 \sqcap \forall R_1.A_1 \sqcap ... \sqcap \forall R_n.A_n$

With axioms of form (I), concept (role) names can be declared to be subconcepts (subroles) of each other. Axioms of form (II) denote disjointness between concepts. Axioms of type (III) introduce domain and range restrictions for roles. Axioms of the form (IV) introduce so-called *functional* restrictions on roles, and axioms of type (V) specify local range restrictions (using value restrictions, see below). With axioms of kind (VI) so-called definitions (with necessary and sufficient conditions) can be specified for concept names found on the lefthand side of the $\equiv$ sign. In the axioms, so-called *concepts* are used. Concepts are concept names or expressions of the form $\top$ (anything), $\bot$ (nothing), $\neg A$ (atomic negation), $(\leq 1\, R)$ (role functionality), $\exists R.\top$ (limited existential restriction), $\forall R.A$ (value restriction) and $(C_1 \sqcap ... \sqcap C_n)$ (concept conjunction).

Knowledge about individuals is represented in the Abox part of Σ. An Abox $\mathcal{A}$ is a set of expressions of the form $A(a)$ or $R(a, b)$ (concept assertions and role assertions, respectively) where A stands for a concept name, R stands for a role name, and a, b stand for individuals. Aboxes can also contain equality ($a = b$) and inequality assertions ($a \neq b$). We say that the unique name assumption (UNA) is applied, if $a \neq b$ is added for all pairs of individuals a and b.

In order to understand the notion of logical entailment, we introduce the semantics of $\mathcal{ALH}_f{}^-$. In DLs such as $\mathcal{ALH}_f{}^-$, the semantics is defined with interpretations $\mathcal{I} = (\triangle^\mathcal{I}, \cdot^\mathcal{I})$, where $\triangle^\mathcal{I}$ is a non-empty set of domain objects (called the domain of $\mathcal{I}$) and $\cdot^\mathcal{I}$ is an interpretation function which maps individuals to objects of the domain ($a^\mathcal{I} \in \triangle^\mathcal{I}$), atomic concepts to subsets of the domain ($A^\mathcal{I} \subseteq \triangle^\mathcal{I}$) and roles to subsets of the cartesian product of the domain ($R^\mathcal{I} \subseteq \triangle^\mathcal{I} \times \triangle^\mathcal{I}$). The interpretation of arbitrary $\mathcal{ALH}_f{}^-$ concepts is then defined by extending $\cdot^\mathcal{I}$ to all $\mathcal{ALH}_f{}^-$ concept constructors as follows:

$$
\begin{array}{lcl}
\top^{\mathcal{I}} & = & \triangle^{\mathcal{I}} \\
\bot^{\mathcal{I}} & = & \emptyset \\
(\neg A)^{\mathcal{I}} & = & \triangle^{\mathcal{I}} \setminus A^{\mathcal{I}} \\
(\leq 1\, R)^{\mathcal{I}} & = & \{u \in \triangle^{\mathcal{I}} \mid (\forall v_1, v_2)\, [((u, v_1) \in R^{\mathcal{I}} \wedge (u, v_2) \in R^{\mathcal{I}}) \rightarrow v_1 = v_2] \\
(\exists R.\top)^{\mathcal{I}} & = & \{u \in \triangle^{\mathcal{I}} \mid (\exists v)\, [(u, v) \in R^{\mathcal{I}}]\} \\
(\forall R.C)^{\mathcal{I}} & = & \{u \in \triangle^{\mathcal{I}} \mid (\forall v)\, [(u, v) \in R^{\mathcal{I}} \rightarrow v \in C^{\mathcal{I}}]\} \\
(C_1 \sqcap ... \sqcap C_n)^{\mathcal{I}} & = & C_1^{\mathcal{I}} \cap ... \cap C_n^{\mathcal{I}}
\end{array}
$$

In the following, the *satisfiability* condition for axioms and assertions of an $\mathcal{ALH}_f{}^-$-knowledge base Σ in an interpretation $\mathcal{I}$ are defined. A concept inclusion $C \sqsubseteq D$ (concept definition $C \equiv D$) is satisfied in $\mathcal{I}$, if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ (resp. $C^{\mathcal{I}} = D^{\mathcal{I}}$) and a role inclusion $R \sqsubseteq S$ (role definition $R \equiv S$), if $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$ (resp. $R^{\mathcal{I}} = S^{\mathcal{I}}$). Similarly, assertions $C(a)$ and $R(a, b)$ are satisfied in $\mathcal{I}$, if $a^{\mathcal{I}} \in C^{\mathcal{I}}$ resp. $(a, b)^{\mathcal{I}} \in R^{\mathcal{I}}$. If an interpretation $\mathcal{I}$ satisfies all axioms of $\mathcal{T}$ resp. $\mathcal{A}$ it is called a *model* of $\mathcal{T}$ resp. $\mathcal{A}$. If it satisfies both $\mathcal{T}$ and $\mathcal{A}$ it is called a model of Σ. Finally, if there is a model of Σ (i.e., a model for $\mathcal{T}$ and $\mathcal{A}$), then Σ is called satisfiable.

We are now able to define the entailment relation $\models$. A DL knowledge base Σ *logically entails* an assertion α (symbolically $\Sigma \models \alpha$) if α is satisfied in all models of Σ. For an Abox $\mathcal{A}$, we say $\Sigma \models \mathcal{A}$ if $\Sigma \models \alpha$ for all $\alpha \in \mathcal{A}$.

2.2 Substitutions, Queries, and Rules

2.2.1 Sequences, Variable Substitutions and Transformations

A *variable* is a name of the form *String* where *String* is a string of characters from {A...Z}. In the following definitions, we denote places where variables can appear with uppercase letters.

Let V be a set of variables, and let $\underline{X}, \underline{Y_1}, \ldots, \underline{Y_n}$ be sequences $\langle \ldots \rangle$ of variables from V. The notation $\underline{z}$ denotes a sequence of individuals. We consider sequences of length 1 or 2 only, if not indicated otherwise, and assume that $(\langle X \rangle)$ is to be read as (X) and $(\langle X, Y \rangle)$ is to be read as (X, Y) etc. Furthermore, we assume that sequences are automatically flattened. A function *as_set* turns a sequence into a set in the obvious way.

A *variable substitution* $\sigma = [X \leftarrow i, Y \leftarrow j, \ldots]$ is a mapping from variables to individuals mentioned in an Abox. The application of a variable substitution σ to a sequence of variables $\langle X \rangle$ or $\langle X, Y \rangle$ is defined as $\langle \sigma(X) \rangle$ or $\langle \sigma(X), \sigma(Y) \rangle$, respectively, with $\sigma(X) = i$ and $\sigma(Y) = j$. In this case, a sequence of individuals is defined. If a substitution is applied to a variable X for which there exists no mapping $X \leftarrow k$ in σ then the result is undefined. A variable for which all required mappings are defined is called *admissible* (w.r.t. the context).

2.2.2 Grounded Conjunctive Queries

Let $\underline{X}, \underline{Y_1}, \ldots, \underline{Y_n}$ be sequences of variables, and let $Q_1, \ldots, Q_n$ denote concept or role names. A query is defined by the following syntax: $\{(\underline{X}) \mid Q_1(\underline{Y_1}), \ldots, Q_n(\underline{Y_n})\}$. The sequence $\underline{X}$ may be of arbitrary length but all variables mentioned in $\underline{X}$ must also appear in at least one of the $\underline{Y_1}, \cdots, \underline{Y_n}$: $as\_set(\underline{X}) \subseteq as\_set(\underline{Y_1}) \cup \cdots \cup as\_set(\underline{Y_n})$.

Informally speaking, $Q_1(\underline{Y_1}), \ldots, Q_n(\underline{Y_n})$ defines a conjunction of so-called *query atoms* $Q_i(\underline{Y_i})$. The list of variables to the left of the sign | is called the *head* and the atoms to the right are called the query *body*. The variables in the head are called distinguished variables. They define the query result. The variables that appear only in the body are called non-distinguished variables and are existentially quantified. Answering a query with respect

to a knowledge base Σ means finding admissible variable substitutions σ such that $\Sigma \models \{\sigma(Q_1(\underline{Y_1})), \ldots, \sigma(Q_n(\underline{Y_n}))\}$. We say that a variable substitution $\sigma = [X \leftarrow i, Y \leftarrow j, \ldots]$ introduces *bindings* $i, j, \ldots$ for variables $X, Y, \ldots$. Given all possible variable substitutions σ, the *result* of a query is defined as $\{(\sigma(\underline{X}))\}$. Note that the variable substitution σ is applied before checking whether $\Sigma \models \{Q_1(\sigma(\underline{Y_1})), \ldots, Q_n(\sigma(\underline{Y_n}))\}$, i.e., the query is *grounded* first.

A *boolean* query is a query with $\underline{X}$ being of length zero. If for a boolean query there exists a variable substitution σ such that $\Sigma \models \{\sigma(Q_1(\underline{Y_1})), \ldots, \sigma(Q_n(\underline{Y_n}))\}$ holds, we say that the query is answered with *true*, otherwise the answer is *false*. Later on, we will have to convert query atoms into Abox assertions. This is done with the function *transform*. The function *transform* applied to a set of query atoms $\{\gamma_1, \ldots \gamma_n\}$ is defined as $\{transform(\gamma_1, \sigma), \ldots, transform(\gamma_n, \sigma)\}$ where $transform(P(\underline{X}), \sigma) := P(\sigma(\underline{X}))$.

2.2.3 Rules

A rule r has the following form $P(\underline{X}) \leftarrow Q_1(\underline{Y_1}), \ldots, Q_n(\underline{Y_n})$ where P, $Q_1, \ldots, Q_n$ denote concept or role names with the additional restriction (safety condition) that $as\_set(\underline{X}) \subseteq as\_set(\underline{Y_1}) \cup \cdots \cup as\_set(\underline{Y_n})$. Rules are used to derive new Abox assertions, and we say that a rule r is *applied* to an Abox $\mathcal{A}$. The function call $apply(\Sigma, P(\underline{X}) \leftarrow Q_1(\underline{Y_1}), \ldots, Q_n(\underline{Y_n}), \mathcal{A})$ returns a set of Abox assertions $\{\sigma(P(\underline{X}))\}$ if there exists an admissible variable substitution σ such that the answer to the query

$$\{() \mid Q_1(\sigma(\underline{Y_1})), \ldots, Q_n(\sigma(\underline{Y_n}))\}$$

is *true* with respect to $\Sigma \cup \mathcal{A}$.[1] If no such σ can be found, the result of the call to $apply(\Sigma, r, \mathcal{A})$ is the empty set. The application of a set of rules $\mathcal{R} = \{r_1, \ldots r_n\}$ to an Abox is defined as follows:

$$apply(\Sigma, \mathcal{R}, \mathcal{A}) = \bigcup_{r \in \mathcal{R}} apply(\Sigma, r, \mathcal{A})$$

The result of $forward\_chain(\Sigma, \mathcal{R}, \mathcal{A})$ is defined to be $\emptyset$ if $apply(\Sigma, \mathcal{R}, \mathcal{A}) \cup \mathcal{A} = \mathcal{A}$ holds. Otherwise the result of *forward_chain* is determined by the recursive call $apply(\Sigma, \mathcal{R}, \mathcal{A}) \cup forward\_chain(\Sigma, \mathcal{R}, \mathcal{A} \cup apply(\Sigma, \mathcal{R}, \mathcal{A}))$.

For some set of rules $\mathcal{R}$ we extend the entailment relation by specifying that $(\mathcal{T}, \mathcal{A}) \models_{\mathcal{R}} \mathcal{A}_0$ iff $(\mathcal{T}, \mathcal{A} \cup forward\_chain((\mathcal{T}, \emptyset), \mathcal{R}, \mathcal{A})) \models \mathcal{A}_0$.

2.3 Probabilistic Formalism

An observation i.e. an assertion which the agent receives has a so called certainty value. This certainty value represents the degree of belief of another agent in that particular assertion. All certainty values are interpreted by the receiving agent as the probability that the corresponding assertion is true. In [2] it was shown how probabilities of observation Aboxes can be computed given the $P(\mathcal{A}, \mathcal{A}', \mathcal{R}, \mathcal{WR}, \mathcal{T})$ function which determines the probability of the observation Abox $\mathcal{A}$ with respect to the Abox $\mathcal{A}'$ containing observations and explanations, a set of rules $\mathcal{R}$, a set of weighted rules $\mathcal{WR}$, and the Tbox $\mathcal{T}$ where $\mathcal{A} \subseteq \mathcal{A}'$. Note that $\mathcal{R}$ is a set of forward and backward chaining rules. The probability determination is performed based on the Markov logic formalism (see [2] for more details).

Henceforth we use $P(\mathcal{A}')$ as an abbreviation for $P(\mathcal{A}, \mathcal{A}', \mathcal{R}, \mathcal{WR}, \mathcal{T})$.

[1] We slightly misuse notation in assuming $(\mathcal{T}, \mathcal{A}) \cup \Delta = (\mathcal{T}, \mathcal{A} \cup \Delta)$. If $\Sigma \cup \mathcal{A}$ is inconsistent the result is well-defined but useless. It will not be used afterwards.

3 The Media Interpretation Agent

This section presents an extended version of the *MI_Agent*, first described in [1] and [2]. The possibility to generate and ask queries for disambiguation was added and is represented by the *generateQueries* function and the *ask* function, respectively. The generation of these queries is described in more detail in Section 3.1. The agent also obtained a second input queue that runs independently in a separate thread and is used exclusively for handling answers to previously asked questions. This newly introduced ability and the way those query responses are communicated is explained in Section 3.2.

Function MI_Agent($\mathcal{Q}_\Gamma$,$\mathcal{Q}_\Upsilon$, *partners*, *die*, $(\mathcal{T}, \mathcal{A}_0), \mathcal{FR}, \mathcal{BR}, \mathcal{WR}, k, \epsilon$)
Input: a queue of observations $\mathcal{Q}_\Gamma$, a queue of responses to queries $\mathcal{Q}_\Upsilon$, a set of partners *partners*, a termination function *die*(), a background knowledge base $(\mathcal{T}, \mathcal{A}_0)$, a set of forward chaining rules $\mathcal{FR}$, a set of backward chaining rules $\mathcal{BR}$, a set of weighted rules $\mathcal{WR}$, a parameter k indicating the top k Aboxes on the agenda, and a threshold control parameter ϵ
Output: –

$currentI = \emptyset$, $\mathfrak{A} = \{\emptyset\}$;
$startThread(\lambda().$
repeat
$\Gamma := extractObservations(\mathcal{Q}_\Gamma)$;
$W := MAP(\Gamma, \mathcal{WR}, \mathcal{T})$;
$\Gamma' := select(W, \Gamma)$;
$\mathfrak{A}' := filter(\lambda(\mathcal{A}).consistent_\Sigma(\mathcal{A}),$
$map(\lambda(\mathcal{A}).\Gamma' \cup \mathcal{A} \cup \mathcal{A}_0 \cup$
$forwardChain(\Sigma, \mathcal{FR}, \Gamma' \cup \mathcal{A} \cup \mathcal{A}_0),$
$\{select(MAP(\Gamma' \cup \mathcal{A} \cup \mathcal{A}_0, \mathcal{WR}, \mathcal{T}),$
$\Gamma' \cup \mathcal{A} \cup \mathcal{A}_0) \mid \mathcal{A} \in \mathfrak{A}\}));$
$(\mathfrak{A}, newI, \Delta^+, \Delta^-) := interpret(\mathfrak{A}', currentI, \Gamma', (\mathcal{T}, \mathcal{A}_0),$
$\mathcal{FR}, \mathcal{BR}, \mathcal{WR} \cup \Gamma, \epsilon);$
$currentI := newI$;
$communicate(\Delta^+, \Delta^-, partners)$;
$\mathfrak{A} := manageAgenda(\mathfrak{A})$;
$Q := generateDisambiguationQuery(\mathfrak{A}, k)$;
if $Q \neq \emptyset$ **then** $ask(Q, partners)$ **end**;
until *die*();
);
$startThread(\lambda().$
repeat
$\Upsilon := extractQueryAnswer(\mathcal{Q}_\Upsilon)$;
$\mathcal{A}_a := find(\Upsilon, \mathfrak{A})$;
$update(\mathcal{A}_a)$;
$\mathfrak{A} := shift(\mathcal{A}_a, \mathfrak{A})$;
$newI := selectHead(\mathfrak{A})$;
$(\Delta^+, \Delta^-) := AboxDiff(newI, currentI)$;
$currentI := newI$;
$communicate(\Delta^+, \Delta^-, partners)$;
until *die*();
);

The MI_Agent uses a set of auxiliary functions, which are defined as follows.

$$\mathit{filter}(f, X) = \bigcup_{x \in X} \begin{cases} \{x\} & \text{if } f(x) = \mathit{true} \\ \emptyset & \text{else} \end{cases}$$

It takes as parameters a function f and a set X and returns a set consisting of the values of f applied to every element x of X. In the MI_Agent function, the current interpretation $currentI$ is initialised to empty set and the agenda $\mathfrak{A}$ to a set containing empty set. Since the agent performs an incremental process, it is defined by repeat-loops. In case the agent receives a percept result Γ, it is sent to the queue $\mathcal{Q}_\Gamma$. In order to take the observations Γ from the queue $\mathcal{Q}_\Gamma$, the MI_Agent calls the $extractObservations$ function. The function $MAP(\Gamma, \mathcal{WR}, \mathcal{T})$ determines the most probable world of observations Γ' with respect to a set of weighted rules $\mathcal{WR}$ and the Tbox $\mathcal{T}$. It returns a vector W which consists of ones and zeros assigned to indicate whether the ground atoms of the considered world are true (positive) or false (negative), respectively. The function $select(W, \Gamma)$ then selects the positive assertions in the input Abox Γ using the bit vector W. The selected positive assertions are the assertions which require explanations. The $select$ operation returns as output an Abox Γ' which has the following characteristic: $\Gamma' \subseteq \Gamma$. The determination of the most probable world by the MAP function and the selection of the positive assertions is carried out on $\Gamma' \cup \mathcal{A} \cup \mathcal{A}_0$. In the next step, a set of forward chaining rules $\mathcal{FR}$ is applied to $\Gamma' \cup \mathcal{A} \cup \mathcal{A}_0$. The generated assertions in this process are added to $\Gamma' \cup \mathcal{A} \cup \mathcal{A}_0$. In the next step, only the consistent Aboxes are selected and the inconsistent Aboxes are removed. Afterwards, the $interpret$ function is called to determine the new agenda $\mathfrak{A}$, the new interpretation Abox $newI$ and the Abox differences Δ^+ and Δ^- for additions and omissions among $currentI$ and $newI$. Afterwards, the Abox $currentI$ is assigned to $newI$ and the MI_Agent function communicates the Abox differences Δ^+ and Δ^- to $partners$. In CASAM $partners = \{KDMA, HCI\}$ applies. Subsequently, the $manageAgenda$ function is called with the aim to improve the overall performance. It is applied to the agenda $\mathfrak{A}$ which usually contains multiple interpretation Aboxes and makes use of the following techniques:

- Elimination of the interpretation Aboxes: This technique is applied if there are multiple interpretation Aboxes with different scoring values where one of the Aboxes has a higher scoring value. At this step, we can select this Abox, eliminate the remaining interpretation Aboxes and continue the interpretation process with the selected Abox.

- Combining the interpretation Aboxes: Consider the interpretation Aboxes $I_1, \ldots, I_n$. In order to determine the final interpretation Abox, the MAP process can be applied to the union of all interpretation Aboxes $I_1 \cup \ldots \cup I_n$. The MAP process determines the most probable world based on the Tbox $\mathcal{T}$ and the set of weighted rules $\mathcal{WR}$.

The last two functions in the processing loop for observations are $generateQueries$ and a conditional ask. As stated above, they are used to calculate disambiguation queries and communicate them to the partners. The former function is explained in more detail in the next Section 3.1.
The termination condition of the MI_Agent function is that the function die is true. Note that the MI_Agent waits in the function call $extractObservations(\mathcal{Q}_\Gamma)$ if $\mathcal{Q}_\Gamma = \emptyset$ and $\mathcal{Q}_\Upsilon = \emptyset$.

3.1 Generation of Disambiguation Queries

As described in the previous section, we consider an agent that receives some percepts in an environment. These percepts typically come from other agents that act in the same environment. Please note that the term agent in this sense is not restricted to machines or programs only; we also want to think of agents that represent the action of humans, or even a direct interaction of humans and agents. This view is adopted in the CASAM project and allows the interaction of the system with a human expert. The very basic functionality of receiving percepts from other agent's is only one part of an agents ability that is often referred to as *social ability*. As a logical consequence, agents are also able to produce output that affects the environment. One type of output are messages that are sent to other participants. To be even more specific, queries can be sent to humans in order to facilitate the interpretation process. The selection of an Abox $\mathcal{A}_i$ from $\mathfrak{A}$ representing the interpretation with the maximum score is an essential step in the *interpret* function. In fact, there could be multiple Aboxes which satisfy this criterion or, in a slightly weakened condition, do not differ much. In this case, the generation of queries for disambiguation between preferable Aboxes is performed by the agent. The function *generateDisambiguationQuery* is defined as follows.

Function generateDisambiguationQuery($\mathfrak{A}$, k)
Input: an agenda $\mathfrak{A}$ and an offset parameter k
Output: a disambiguation query Q
$\mathfrak{I} := \{\mathcal{A}_i \in \mathfrak{A} \mid i = 1 \dots k\}$;
$\mathcal{D} := \emptyset$, $Q := \emptyset$;
foreach $\mathcal{A}_i \in \mathfrak{I}$ **do**
  $\mathcal{D}_{\mathcal{A}_i} := \bigcap_{1 \leq j \leq k, i \neq j} \mathcal{A}_i \setminus \mathcal{A}_j$;
  $d_{\mathcal{A}_i} := selectAssertion(\mathcal{D}_{\mathcal{A}_i})$;
  $\mathcal{D} := \mathcal{D} \cup d_{\mathcal{A}_i}$;
end
if $\mathcal{D} \neq \emptyset$ **then**
  $Op := computeLogicalCompound(\mathcal{D})$;
  $Iv := computeImportanceValue(\mathcal{D}, \mathfrak{I})$;
  $Q := concat(buildQuery(\mathcal{D}, Op), Iv)$;
end
return Q;

It takes as input an agenda $\mathfrak{A}$ and an offset parameter k. Since the Aboxes $\mathcal{A}_i$ on $\mathfrak{A}$ are ordered by a scoring function, the integer value k is used to determine the top k Aboxes on the Agenda, respectively those k Aboxes with the highest scores, denoted as $\mathfrak{I}$. Additionally, a set of difference assertions $\mathcal{D}$ is initialised to empty set. After the Aboxes were selected from $\mathfrak{A}$, each of them is processed in a foreach loop. First, the set $\mathcal{D}_{\mathcal{A}_i}$, denoting the intersection of all Abox differences between the currently chosen Abox $\mathcal{A}_i$ and all other Aboxes $\mathcal{A}_j, 1 \leq j \leq k$ and $i \neq j$, is computed.
Compared to all other Aboxes, these assertions are unique to $\mathcal{A}_i$. Considering Aboxes as sets, Figure 3 shows a schematic diagram of the Abox difference operation. Aboxes are represented by circles and intersections of them are marked as darker grey areas. Assuming the Abox $\mathcal{A}_i$ is the current selected Abox and the area $\mathcal{D}_{\mathcal{A}_i}$ represents the assertion that is unique to $\mathcal{A}_i$. If this set contains more than one assertion, one of them is chosen randomly, because it does not matter which assertion represents the uniqueness. This step is realised by the function *selectAssertion*. The resulting assertion $d_{\mathcal{A}_i}$ is then added to $\mathcal{D}$.

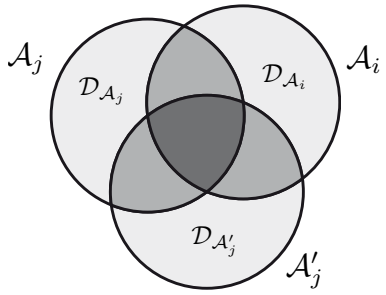

Figure 3: Schema of Abox difference

After the loop terminates, $\mathcal{D} = \{d_{\mathcal{A}_1}, ..., d_{\mathcal{A}_k}\}$ holds difference assertions that are unique to every Abox among the top k Aboxes on the agenda $\mathfrak{A}$. Subsequently, the found difference assertions must be transformed into a query. This is done by using a logical conjunction. The logical compound that is told by the agent to connect the assertions is **OR** if there is no pair of assertions that is inconsistent with the background knowledge and **XOR** otherwise. This condition is examined by the *computeLogicalCompound* function and the result is assigned to an operator variable Op. The next step includes the calculation of an importance value Iv, for which a first proposal is given at the end of this section.

Function buildQuery($\mathcal{D}$, Op)
Input: a set of assertions $\mathcal{D}$ and a logical operator Op
Output: assertions concatenated with a logical operator
if $\mathcal{D} = \{d_{\mathcal{A}_i}\}$ **then**
    return $d_{\mathcal{A}_i}$;
else
    return $concat(d_{\mathcal{A}_i}, Op, buildQuery(\mathcal{D} \setminus d_{\mathcal{A}_i}, Op))$;
end

Afterwards, the query Q is built by using the recursive function *buildQuery* that takes as input all the difference assertions, hold in $\mathcal{D}$, together with the logical operator Op. As a result, all assertions, concatenated with Op, are returned. This concatenated construct, together with the calculated value Iv, builds the final query that is communicated to the partners. Syntactically, we define a *disambiguation query*, or query for short, using the Backus-Naur form as

$$
\begin{aligned}
Q &::= Q'\ Iv \mid Q''\ Iv \\
Q' &::= \alpha \mid \alpha\ \mathbf{OR}\ Q' \\
Q'' &::= \alpha \mid \alpha\ \mathbf{XOR}\ Q''
\end{aligned}
$$

where α is an assertion, and $Iv \in (0, 1]$ is a real value denoting the so-called importance value of the query.

Importance Value

Queries are asked to users in order to reduce the large space of abducibles to the most appropriate explanations for the observations. In order to specify the degree of disambiguation that is expected by RMI when a particular query is answered, each generated

query is associated with an importance value. In the following, a first proposal is presented to compute these values. For answering queries, the user has to invest an amount of time as well as some cognitive ressoures ("costs") such that this proposal is not based on simply considering the number of query disjuncts that are asked to the user.

In order to obtain a degree of disambiguation, queries can be favoured in which there is a pair of disjuncts $(d_{\mathcal{A}_i}, d_{\mathcal{A}_j})$ where the probabilities of the corresponding Aboxes $(\mathcal{A}_i, \mathcal{A}_j)$ are most similar. According to this, an importance value is the reciprocal of the minimum $min(|P(\mathcal{A}_i) - P(\mathcal{A}_j)|)$ of all pairs of disjuncts.

However, in addition, queries can be ranked according to the similarity of the disjuncts itself, since the more similar the disjuncts are, the more they cannot hold in parallel (i.e., according to the current scene in the video), and, following to this, the more disjuncts probably will be disambiguated by the user. For example, the concepts *Drought* and *Pollution* are less probable to hold in parallel than *Drought* and *Microphone* (being less similar) such that an answer to the former pair is believed to provide more information for disambiguation than an answer to the latter pair. An estimation for the similarity of a pair of disjuncts is the reciprocal of their distance with respect to the underlying taxonomy. For computing the taxonomical distance of a pair of disjuncts (A_i, A_j), the least common subsumer $lcs(A_i, A_j)$ [7] of these concepts is computed. Then, $dist(A_i, A_j)$ is the sum of the distances of A_i to $lcs(A_i, A_j)$ and A_j to $lcs(A_i, A_j)$ (cf. [8]). In the case that $A_i = lcs(A_i, A_j)$ resp. $A_j = lcs(A_i, A_j)$ one of the disjuncts subsumes the other such that no disambiguation is needed and $dist(A_i, A_j)$ is set to the maximal taxonomic distance of pairs of disjuncts occurring (or to the maximal depth of the underlying taxonomy, if there is no such maximum). The same holds for queries with a single disjunct, for pairs of query disjuncts that include a role, or for disjuncts that refer to the same concept, i.e., differ only in the associated individual.

Let $pairs$ be the set of all unordered pairs $(d_{\mathcal{A}_i}, d_{\mathcal{A}_j})$ of query disjuncts. An importance value for a set $\mathcal{D}$ of query disjuncts in $(0, 1]$ is defined by

$$importance(\mathcal{D}) = \frac{1}{(1+min(|P(\mathcal{A}_i)-P(\mathcal{A}_j)|))\cdot\frac{1}{2|pairs|}\sum_{pairs} dist(pairs)}.$$

The minimal distance of two concepts is 2, since pairs of disjuncts subsuming each other are excluded. In order to guarantee that the maximal importance of a query is 1, the sum of distances is divided by $2\,|pairs|$.

3.2 Processing of Query Responses

In the beginning of this section the updated *MI_Agent* was presented. As mentioned before, the agent now makes use of two distinct threads. In the following we present how the agent reacts to user responses, buffered in the queue $\mathcal{Q}_\Upsilon$. Each assertion in $\mathcal{D} = \{d_{\mathcal{A}_1}, ..., d_{\mathcal{A}_i}\}$, computed by the *generateQueries* function, has also a certainty value c_i. This certainty value represents how confident the agent is about this assertion. After presenting the queries in an human readable form on the user interface, the user is able to select one or more assertions as an answer. A confirmation of an assertion results in a response where the certainty is increased to 1 and a rejection of the user, given by a non selected answer possibility, lowers the certainty value to 0.

As soon as the agent receives a response, it extracts the query answer from $\mathcal{Q}_\Upsilon$ by calling the function *extractQueryAnswer*$(\mathcal{Q}_\Upsilon)$. Afterwards it has to look up all the Aboxes which contain the assertions in the response. This functionality is provided by the function *find*$(\Upsilon, \mathfrak{A})$ which returns a set of all interpretation Aboxes $\mathfrak{I}_a = \{\mathcal{A}_a \in \mathfrak{A} \mid \upsilon \in \mathcal{A}_a \wedge \upsilon \in \Upsilon\}$ that are affected by the answer. Then these Aboxes have to be updated according to the

new certainty value of these assertions given by the answer from the user. This is done by the function *update*($\mathfrak{I}_a$). Changes of the certainty value of a particular assertion can have an influence on the certainty value of structures built upon that assertion as well as on the probability of the whole Abox. Therefore, the certainty values of the assertions and the probability of the Abox has to be recomputed. If the probability of an Abox changes, also the ranking of that Abox in the agenda might have changed and the position has to be updated. This is done by the function *shift*($\mathfrak{I}_a, \mathfrak{A}$) which sorts all Aboxes from $\mathfrak{I}_a$ such that $\{\mathcal{A}_1, \ \dots \ , \mathcal{A}_i, \mathcal{A}_a, \mathcal{A}_j, \ \dots \ , \mathcal{A}_n \mid P(\mathcal{A}_{\leq i}) \geq P(\mathcal{A}_a) \geq P(\mathcal{A}_{\geq j})\}$ holds. As a consequence of the *shift* operation the most probable Abox, ranked at the first position of the agenda, might have changed. To handle this possibility, the eventually new most probable Abox is selected by *selectHead* and assigned to *newI*. Afterwards, the Abox-Difference (see [1]) between this Abox and the former best Abox *currentI* (and vice versa) is computed by *AboxDiff*. This operation results in additions Δ^+ as well as in omissions Δ^-. The next step consists of assigning the newly found most probable Abox to *currentI*. Finally, the additions and omissions are communicated to the partners.

4 Query Types

Queries can be further classified in different types depending on the disjuncts they include. First, queries can contain concept assertions resp. role assertions only. Further, the disjuncts can be represent concrete or abstract knowledge or the order of individuals in role assertions can differ. These types are required for the human-computer-interaction component (HCI) in order to display the respective query accordingly. In the following, the types that have been agreed on are presented in the form of examples including figures that sketch the presentation of these example queries on the screen. As we focus on the types of the disjuncts in this section, importance values are not considered for these examples. For further details, background regarding the graphical user interface and several context-specific screenshots, please refer to [9].

4.1 Concept-Assertion-Query

Let us assume the RMI agent receives the following observations:

$$\mathcal{A}_0 = \{PersonName(IND_0), hasConcreteValue(IND_0, \text{“Bob”})\}$$

Based on the observations and the background knowledge the following interpretations are provided by the agent.

$$\mathcal{I}_0 = \{EnergyMinister(IND_1), hasName(IND_1, IND_0)\}$$
$$\mathcal{I}_1 = \{Journalist(IND_1), hasName(IND_1, IND_0)\}$$
$$\mathcal{I}_2 = \{PopStar(IND_1), hasName(IND_1, IND_0)\}$$

If the concepts $EnergyMinister$, $Journalist$ and $Popstar$ are disjoint with respect to the background knowledge, RMI generates an **XOR**-query, since each concept assertion is associated to the same individual IND_1:[2]

$$Q'_1 = EnergyMinister(IND_1) \ \mathbf{XOR} \ Journalist(IND_1) \ \mathbf{XOR} \ PopStar(IND_1)$$

[2]Please note that the format of the query presented here is not the format transmitted through the CASAM system. For exact XML-schema specifications refer to [10]

If *EnergyMinister*, *Journalist* and *PopStar* are not specified pairwise disjoint, RMI generates an **OR**-query:

$$Q'_2 = EnergyMinister(IND_1) \textbf{ OR } Journalist(IND_1) \textbf{ OR } PopStar(IND_1)$$

The **OR**-query Q'_2 is shown on the UI in the form of a natural language question with a check box for each concept as depicted in Figure 4. By activating a check box a users confirms the associated assertion computed by the RMI agent, whereby leaving a check box inactivated means an objection of the assertion. This applies to all **OR**-queries.

Is "Bob" a(n)...
☑ Energy minister?
☐ Journalist?
☐ Pop star?

Figure 4: Example of a Concept-Assertion-Query

In the case of **XOR**-queries, the UI provides radio buttons such that only a single choice is possible.

4.2 Abstract-Concept-Assertion-Query

Concept assertions in queries may refer to individuals which are neither related to concrete values nor to bounding boxes. For example, given the observations

$$\mathcal{A}_1 = \{Winds(IND_2), Power(IND_3)\}$$

and assumed that RMI has computed the following two interpretations:

$\mathcal{I}_3 = \{WindEnergy(IND_4), associatedWith(IND_4, IND_2), associatedWith(IND_4, IND_3)\}$
$\mathcal{I}_4 = \{Climate(IND_4), associatedWith(IND_4, IND_2), associatedWith(IND_4, IND_3)\}$

the resulting query is similar to Q'_2:

$$Q'_3 = WindEnergy(IND_4) \textbf{ OR } Climate(IND_4)$$

As a result HCI will display the query as depicted in Figure 5.

Tick all that apply:
☑ Wind energy
☐ Climate change

Figure 5: Example of an Abstract-Concept-Assertion-Query

4.3 Relation-Direction-Query

Queries may not only refer to concept assertions but also to relation assertions. For example, RMI can ask for the order of the individuals of a specific role assertion. Given the observations

$$\mathcal{A}_2 = \{PersonName(IND_4), hasConcreteValue(IND_4, \text{“John”}),\\ PersonName(IND_5), hasConcreteValue(IND_5, \text{“Pete”})\},$$

we assume that the following interpretations are given by the RMI agent:

$$\begin{aligned}\mathcal{I}_5 = \{&Person(IND_5), associatedWith(IND_5, IND_4),\\ &Person(IND_6), associatedWith(IND_6, IND_5), visits(IND_6, IND_5)\}\\ \mathcal{I}_6 = \{&Person(IND_5), associatedWith(IND_5, IND_4),\\ &Person(IND_6), associatedWith(IND_6, IND_5), visits(IND_5, IND_6)\}\end{aligned}$$

The only difference between $\mathcal{I}_5$ and $\mathcal{I}_6$ is the order of the individuals which are related with the role *visits*. Since the intention is that only one assertion can be true, RMI generates an **XOR**-query:

$$Q'_4 = visits(IND_5, IND_6) \textbf{ XOR } visits(IND_6, IND_5)$$

HCI displays the **OR**-query as shown in Figure 6. In this case the user can only confirm one of the possibilities and all other are automatically rejected. This applies to all **XOR**-queries.

Choose which applies:
○ John visits Pete
⊙ Pete visits John

Figure 6: Example of a Relation-Direction-Query

4.4 Relation-Object-Query

A Relation-Object-Query is a relation query to disambiguate an uncertain object of a relation. It will be sent if two or more interpretations differ in the object of a role assertion such that only the object will vary in this query type.
Given the observations

$$\begin{aligned}\mathcal{A}_3 = \{&CityName(IND_7), hasConcreteValue(IND_7, \text{“Berlin”}),\\ &CityName(IND_8), hasConcreteValue(IND_8, \text{“London”})\}\end{aligned}$$

and assuming that RMI computes the following interpretations:

$$\begin{aligned}\mathcal{I}_7 &= \{Conference(IND_9), associatedWith(IND_9, IND_7), hasLocation(IND_9, IND_7)\}\\ \mathcal{I}_8 &= \{Conference(IND_9), associatedWith(IND_9, IND_8), hasLocation(IND_9, IND_8)\}\end{aligned}$$

The difference results in the following query.

$$Q'_5 = hasLocation(IND_9, IND_7) \textbf{ XOR } hasLocation(IND_9, IND_8)$$

On the UI the query will be presented as depicted in Figure 7.

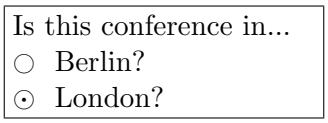

Figure 7: Example of a Relation-Object-Query

4.5 Relation-Subject-Query

A Relation-Subject-Query is a relation query to disambiguate an uncertain subject of a relation. It will be sent if two or more interpretations differ in the subject of a relation. Only the subject will vary in this query type.
Given the observations

$$\begin{aligned}\mathcal{A}_4 = \{&PersonName(IND_{10}), hasConcreteValue(IND_{10}, \text{“John”}),\\ &PersonName(IND_{11}), hasConcreteValue(IND_{11}, \text{“Pete”})\}\end{aligned}$$

RMI computes the following interpretations.

$$\begin{aligned}\mathcal{I}_9 = \{&Person(IND_{12}), associatedWith(IND_{12}, IND_{11}), hasName(IND_{12}, IND_{11})\\ &Person(IND_{13}), associatedWith(IND_{13}, IND_{10}), hasName(IND_{13}, IND_{10})\\ &Conference(IND_{14}), talksAt(IND_{12}, IND_{14})\}\\ \mathcal{I}_{10} = \{&Person(IND_{12}), associatedWith(IND_{12}, IND_{11}), hasName(IND_{12}, IND_{11})\\ &Person(IND_{13}), associatedWith(IND_{13}, IND_{10}), hasName(IND_{13}, IND_{10})\\ &Conference(IND_{14}), talksAt(IND_{13}, IND_{14})\}\end{aligned}$$

The difference results in the following query:

$$Q'_6 = talksAt(IND_{12}, IND_{14}) \textbf{ OR } talksAt(IND_{13}, IND_{14})$$

On the UI the query is shown as can be seen in Figure 8.

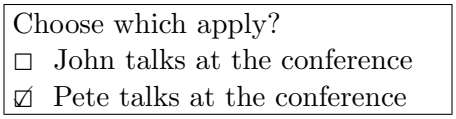

Figure 8: Example of a Relation-Subject-Query

4.6 Relation-Type-Query

A Relation-Type-Query is a relation query to disambiguate an uncertain relation between two individuals. It will be sent if two or more interpretations differ in the relation between two individuals. Only the relation name will vary in this query type. Given the observations

$$\mathcal{A}_5 = \{PersonName(IND_{15}), hasConcreteValue(IND_{15}, \text{“John”})\}$$

The following interpretations are made by the RMI Agent.

$$\mathcal{I}_{11} = \{Person(IND_{16}), associatedWith(IND_{16}, IND_{15}), hasName(IND_{16}, IND_{15}), Car(IND_{17}), enters(IND_{16}, IND_{17})\}$$
$$\mathcal{I}_{12} = \{Person(IND_{16}), associatedWith(IND_{16}, IND_{15}), hasName(IND_{16}, IND_{15}), Car(IND_{17}), exits(IND_{16}, IND_{17})\}$$

The two interpretations differ only in the name of the relation.

$$Q'_7 = enters(IND_{16}, IND_{17}) \textbf{ XOR } exits(IND_{16}, IND_{17})$$

This is displayed on the UI in the form:

Choose which apply?
○ John enters a car
⊙ John exits a car

Figure 9: Example of a Relation-Type-Query

4.7 Relation-Subject-Object-Query

This is the most general form of a query that could be sent. These queries will only be in the form of **OR** statements. Given the observations

$$\mathcal{A}_6 = \{PersonName(IND_{18}), hasConcreteValue(IND_{18}, \text{“John”}) \; LocationName(IND_{19}), hasConcreteValue(IND_{19}, \text{“Hollywood”})\}$$

The following interpretations are made by the RMI Agent.

$$\mathcal{I}_{13} = \{EnergyMinister(IND_{20}), associatedWith(IND_{20}, IND_{18}), hasName(IND_{20}, IND_{18}) \; Location(IND_{21}), associatedWith(IND_{21}, IND_{19}), hasName(IND_{21}, IND_{19})\}$$
$$\mathcal{I}_{14} = \{PopStar(IND_{22}), livesIn(IND_{22}, IND_{21}) \; Location(IND_{21}), associatedWith(IND_{21}, IND_{19}), hasName(IND_{21}, IND_{19})\}$$

The difference is now in two complete different assertions, which leads to the query:

$$Q'_8 = EnergyMinister(IND_{20}) \textbf{ OR } livesIn(IND_{22}, IND_{21})$$

This is displayed on the UI as shown in Figure 10:

Choose which apply?
☐ The energy minister has the name John
☑ The pop star lives in Hollywood

Figure 10: Example of a Relation-Subject-Object-Query

4.8 Same-As-Query

A Same-As-Query is used to disambiguate if two individuals in different modalities refer to the same object in the domain. It will be send if there is an indication for fusion of the two individuals but the evidence leads to a low certainty. Same-As-Queries will be addressed if time permits.

5 Summary and Remarks

At the beginning of this deliverable, three challenges have been posed. This section refers to these challenges and summarises the main results that were achieved.

Generation of queries, including the transformation of relational structures of explanations into flat representations. In Section 3 an enhanced version of the *MI_Agent*, introduced in [1] and [2], was represented. It was shown how it had to be extended in order to facilitate the generation and communication of queries, used to reduce the large space of abducibles. Section 3.1 explained in detail how these queries are computed by the *generateQueries* function based on semantic differences between Aboxes and how a unique assertion for each of them can be found. It was also mentioned that there are two different types of queries, namely **OR**- and **XOR**-queries, that could be generated and asked by the agent.

In the CASAM project a set of query types was agreed upon between the partners. Those types were listed in Section 4. In total eight of them were specified from which the Relation-Subject-Object-Query, exemplarily explained in 4.7, represents the most general form of a query. All other types listed in this deliverable can be seen as special cases.

Computation of importance values for queries. Responses of users to queries have the objective to disambiguate different interpretation possibilities. Section 3.1 presents an approach to the computation of importance values for queries. The general idea behind this value is to prefer queries in which there is a pair of disjuncts where the probabilities of the corresponding Aboxes are most similar. It is also based on the computation of a taxonomical distance, further described in that section.

Processing of answers to queries. This challenge was addressed in Section 3.2 which describes the additions to the former *MI_Agent* that enable it to handle query responses. Therefore, a second thread was established which buffers all incoming answers and processes them with respect to the Aboxes on the agenda. It is also explained which influence those answers have on the agenda.

References

[1] Gries, O., Möller, R., Nafissi, A., Rosenfeld, M., Sokolski, K., Wessel, M.: D3.3 Probabilistic abduction engine: Report on algorithms and the optimization techniques used in the implementation. Project deliverable, Hamburg University of Technology (2010)

[2] Gries, O., Möller, R., Nafissi, A., Rosenfeld, M., Sokolski, K., Wessel, M.: A probabilistic abduction engine for media interpretation. In Alferes, J., Hitzler, P., Lukasiewicz, T., eds.: Proc. International Conference on Web Reasoning and Rule Systems (RR-2010). (2010)

[3] Davis, R.: Meta-rules: Reasoning about control. Artif. Intell. **15**(1) (1980) 179–222

[4] Costantini, S.: Meta-reasoning: A Survey. In: Computational Logic: Logic Programming and Beyond. Springer (2002)

[5] Gries, O., Möller, R., Nafissi, A., Rosenfeld, M., Sokolski, K.: D3.2 Basic reasoning engine: Report on optimization techniques for first-order probabilistic reasoning. Project deliverable, Hamburg University of Technology (2009)

[6] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F., eds.: The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2003)

[7] Cohen, W.W., Borgida, A., Hirsh, H.: Computing least common subsumers in description logics. In: Proc. of AAAI. (1992)

[8] Wolter, K., Smialek, M., Hotz, L., Knab, S., Bojarski, J., Nowakowski, W.: Mapping MOF-based requirements representations to ontologies for software reuse. In: Proc of the 2nd International Workshop on Transformation and weaving ontologies in model driven engineering. (2009)

[9] University of Birmingham: D5.3: HCI final prototype for user studies and evaluation. Project deliverable (2010)

[10] INTRASOFT International S.A.: D7.1: Integrated system conceptual architecture and design (2009)

Annex

A Installation of RMI

The RMI software module (RMI), previously written in Java, was completely rewritten by using the programming language Common Lisp (CL). It comes precompiled as a single archive file for computers running a Linux64 operating system. It is extensively tested under Ubuntu 10.04.1 LTS, but other operating systems, such as Unix, Windows, and Mac OS X are also supported by the CL compiler (although this functionality was not tested during the project). To install RMI, you should have access to the file `RMI.tar.gz` (download link: `http://www.sts.tu-harburg.de/~r.f.moeller/RMI.tar.gz`) and follow the step-by-step instruction below.

- Copy `RMI.tar.gz` to the home directory of the user who installs RMI.
- Gunzip and untar the file `RMI.tar.gz` by using the command:
  `gzip -dc RMI.tar.gz | tar -xvf -`
- Copy the file `~/RMI/hosts.cl` to `~/hosts.cl`
- Copy the file `~/RMI/rmi-master-license.lic` to `~/rmi-master-license.lic`
- In the file `~/RMI/init.racer` you need to specify the domain addresses and ports of the RMI server (see the variable `cl-user::*server-service-url*`) and the Orchestrator (see the variable `cl-user::*client-service-url*`).
- Run the shell script `~/RMI/startup.sh` in order to start the RMI web service. The script is written in a way such that RMI persists if the shell is detached.

The WSDL files on which RMI is based are included into the archive.