Technische Universität Hamburg-Harburg

Query Generation for High-Level Interpretation of Multimedia Documents

Student Project Work

March 2010

submitted by:
Karsten Martiny

supervised by:
Prof. Dr. Ralf Möller
Dipl.-Ing. Maurice Rosenfeld

Technische Universität Hamburg-Harburg
Institute for Software Systems
Schwarzenbergstraße 95
21073 Hamburg

Statement of authorship

I hereby certify that this thesis has been composed by myself and describes my own work, unless otherwise acknowledged in the text. All references and verbatim extracts have been quoted, and all sources of information have been specifically acknowledged. It has not been accepted in any previous application for a degree.

Hamburg,
March 29, 2010
Karsten Martiny

Contents

1 Introduction

In modern multimedia archives there is a huge variety of different multimedia objects (e. g. videos, audio clips, photos). There is a wealth of multimedia databases with an incredibly high potential that currently is barely exploited at all due to the lack of efficient annotation.

In order to exploit the full potential of the available multimedia files, it is crucial to have searchable annotations. Today there are two different approaches to generate these annotations:

One option is to let humans perform the job of annotating the resources. Although they have the means to produce high quality annotations, this approach has a major drawback: A well-done human annotation of a multimedia file is very time-consuming and so the manual approach will either lead to extremely high annotation costs or it will generate a rather unreliable set of annotations. Also, if users take on the job of annotation, each user might have different ideas for categorizing the content and use a variety of spelling, which consequently leads to a much more difficult retrieval process.

The other option available today is to solely rely on machine intelligence for the annotation task. Compared to the manual approach, this method does not require high costs for the annotation. However, current approaches for a machine-only annotation are not mature enough to produce an annotation database with the desired quality.

Based on this situation the project **CASAM** *(Computer-Aided Semantic Annotation of Multimedia)* aims to provide a solution to this problem by facilitating the *synergy of human and machine intelligence* in order to significantly raise the speed and quality of multimedia content annotation. This will be achieved by an annotation tool based on Description Logics that will feature a human-machine interaction loop targeting a fast convergence of human and machine knowledge.

The aproach of a human-machine interaction loop gives rise to the challenge of establishing an efficient and ergonomic communication protocol among participating modules: machine-produced knowledge has to be communicated towards the user in a comprehensible form and requests for additional information should be generated in a user-friendly way. In order to illustrate the communication requirements, this work introduces a set of exemplary scenarios and discusses their courses of interpration in detail. The identification of requirements for the kind of additional information to be obtained from the user emerges from this discussion. Based on those findings, this work proposes feasible syntax and semantics for the generation of queries, as well as strategies for handling these queries in order to enable efficient processing.

This work is structured into five chapters:

After this introduction, the second chapter starts with a general overview of the CASAM project and a description of the approach pursued in this project. Also, that chapter contains a more detailed description of the *Reasoning for Multimedia Interpretation* component (RMI) used for the abduction of additional knowledge. This description focuses especially on CASAM's abduction engine within RMI, since this work's setting is determined by the abduction engine's functionality.

In chapter three, the structure of multimedia documents is introduced, as well as the targeted domain of interpretation scenarios. In succession, this chapter describes a set of sample scenarios from the modeled domain, specifies appropriate rules for their interpretation, and provides a detailed depiction of their respective interpretation courses. Next to a practical demonstration of the procedures described in chapter two, these examples

identify the situations in which queries are created and hereby introduce first hints on the queries' requirements.

Chapter four analyses the information provided by RMI's reasoner, as well as the information required at the user interface. Based upon this analysis, different approaches for the knowledge communication to the user interface are discussed. After determining a suitable approach, this chapter examines requirements for the query's content and proposes an appropriate format. Finally, schemes for query prioritizing are discussed, which allow for an ergonomic and efficent way of handling the queries.

A summary of this work's results is given in chapter five, as well as an outlook for future work.

2 About CASAM

2.1 Project overview

The CASAM *(Computer Aided Semantic Annotation of Multimedia)* project is being developed by a multinational consortium of commercial, academic, and non-profit partners. The group consists of three research institutes, two software companies and three media organizations representing the prospective users of the system. CASAM is co-funded by the European Union under the Seventh Framework Programme for Research (FP7) and supervised by the European Commission's Information Society and Media Directorate-General.[1]

The main objective of CASAM is the development of an annotation tool that facilitates the synergy of human and computer intelligence in order to significantly accelerate the annotation process of multimedia content.

While the project's initial scope will be the modeled domain of news production of News Agencies and Broadcasters, the developed methods aim for a general application on any kind of multimedia content databases.

In order to understand the need for such a semiautomatic tool, it is helpful to have a brief look at the two approaches for the process of multimedia content annotation available today:

1. **Manual Annotation:** Currently this is the standard approach. The job of tagging each content item individually is performed by humans entirely.

   This job can be performed by specialists (for example archivists and librarians). Although this option usually yields to high-quality annotations, there are obviously major drawbacks: since each item is tagged individually by a specialist, the annotation process takes a lot of time and expertise and thus is very cost-intensive. Also, since the result of this process is strictly limited by the number and performance of the available specialists, this approach is only suitable for rather small content databases.

   The other option for manual annotation is the community approach (social tagging like Delicious or Flickr). This approach has the potential to overcome some of the limitations of the specialist's approach: since the annotation process is performed by a (usually large and voluntarily working) crowd it has the potential to annotate large amounts of content without requiring high costs. On the other hand, since there are no hard requirements for the quality and quantity of annotations, the result may vary on a large scale. Also, this introduces a new problem: each individual user may have a different opinion on categories or spelling for the same tag and thereby new obstacles for the retrieval process are being created.

2. **Automatic Annotation:** With this approach, the analysis and annotation of the content items is performed by computers entirely, relying solely on machine intelligence. Since today's technology is still not sophisticated enough to produce optimal annotation results autonomously, this approach cannot yield the desired results:

[1] www.casam-project.eu

If the annotations were limited to what a computer is able to produce with a reasonable error rate, the annotations would be limited to low level content features. Although this approach might create annotations of a certain quality, there will be a huge difference between what a user is able to see with a high level interpretation of the content and what a computer is able to recognize with a low level analysis and thus this approach leads to a so-called *semantic gap*.

It might appear desirable to let the computer perform the high level analysis, also, but since today's technology in this area is very limited, this approach would be highly error prone.

CASAM aims to overcome the mentioned obstacles in the annotation processes by combining these two approaches: CASAM will use all available sources to infer as much information as possible. Its structure (called *Ontology*) is based on a set of concepts and their relations to each other. It uses human input on a case-by-case basis to learn by itself and to expand and upgrade its ontology. It adds new concepts and relations when needed and is thus becoming increasingly "intelligent", i. e. it does not require human interaction for the analysis process once it has gained knowledge from similar cases.

2.2 The approach of CASAM

The project is divided into three separate modules:

- **KDMA** *(Knowledge-Driven Multimedia Analysis)*
  a knowledge database and analysis tool that looks directly at the content in order to retrieve low-level assertions,
- **RMI** *(Reasoning for Multimedia Interpretation)*
  a module that processes the information derived from the initial knowledge-based analysis, tries to retrieve high-level explanations for the obtained observations, asks for user input where necessary, and integrates that input into the ontology as well as the annotation, and
- **HCI** *(Human-Computer Interface)*
  the user interface.

As described in [CAS], these modules form a human-machine interaction loop that will lead to a highly accelerated annotation process:

1. KDMA analyzes the multimedia content (video, image, natural language) and extracts the low level information. This information consists of basic image characteristics such as key objects, presence and number of people, context identification etc., as well as speech recognition and simple concepts derived from text processing.
2. At the same time, the user enters a small number of keywords that describe the high-level concepts present in the multimedia content. Note that the keywords need not belong to a predefined set.
3. RMI augments the KDMA-derived and user-provided information, instantiating appropriately an ontology. Moreover, RMI infers new concept instances and reassesses the context and previous input from KDMA. Results are then fed back to KDMA for multimedia analysis driven by the renewed information. This RMI-KDMA internal information exchange loop continues until nothing more can be inferred by RMI or recognized by KDMA.

4. RMI reasons about what information is needed in order to add missing instances to the ontology or resolve any ambiguities that have arisen in the previous step. If the annotation target has been achieved, the loop exits, otherwise the information requirements are fed to HCI.

5. HCI transforms the information requirements into input requests towards the user. HCI optimizes the user interaction using an effort-cost model, possibly by augmenting requirements to a single input request, using user-modelling to adapt to user information input patterns, etc. Furthermore, the user interface provides the user with the opportunity to alter the knowledge acquisition path devised by the system.

6. HCI input is transferred to RMI and the loop continues from step 3.

A visualization of the interaction loop is depicted in the following figure:

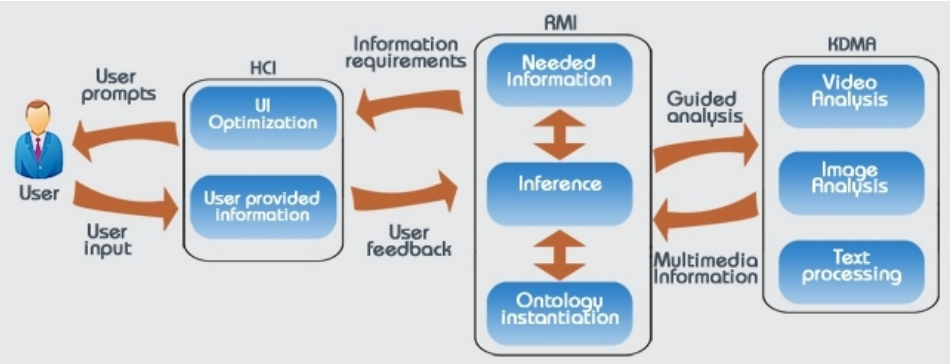


Figure 2.1: Interaction loop between the user and the CASAM modules (from [CAS])

2.3 Architecture of the RMI module

Since this work contributes to CASAM's RMI module, this module is introduced in more detail. For a comprehensive description of the RMI component see [GMN+09].

As described in the previous section, the initial input for RMI consists of low-level content information extracted by KDMA, as well as a set of keywords provided by the user via HCI. This input is initially transformed into an ABox.[2]

In Addition to this content-specific input, the RMI component has background knowledge (a TBox and a set of rules) at its disposal. The signature of this knowledge base contains appropriate names required for representing the knowledge of the domain used by CASAM (edo, *Environmental Domain Ontology*), namely environmental issues and political interviews, as well as names required for representing information about document structures (mco, *Media Content Ontology*). Also, the TBox contains appropriate axioms to relate these names to one another.

After the input assertions received from KDMA and HCI have been transformed into an ABox,[3] this ABox is being processed by three subsequent units as described in [GMN+09]:

- **World Generator:** Based on the TBox and the preprocessed input ABox this component produces all Markov logic worlds, (indicated in 2.2 by $w_1, w_2, ..., w_n$). A

[2] It is assumed that the reader is familiar with the fundamentals of knowledge representation and Descriptions Logics. A comprehensive guide can be found in [BCM+07], for instance.

[3] In fact, some supplementary preprocessing for performance enhancements is performed on this ABox before RMI actually starts processing the data. However, since this preprocessing step does not influence the procedure's results, it is not described here.

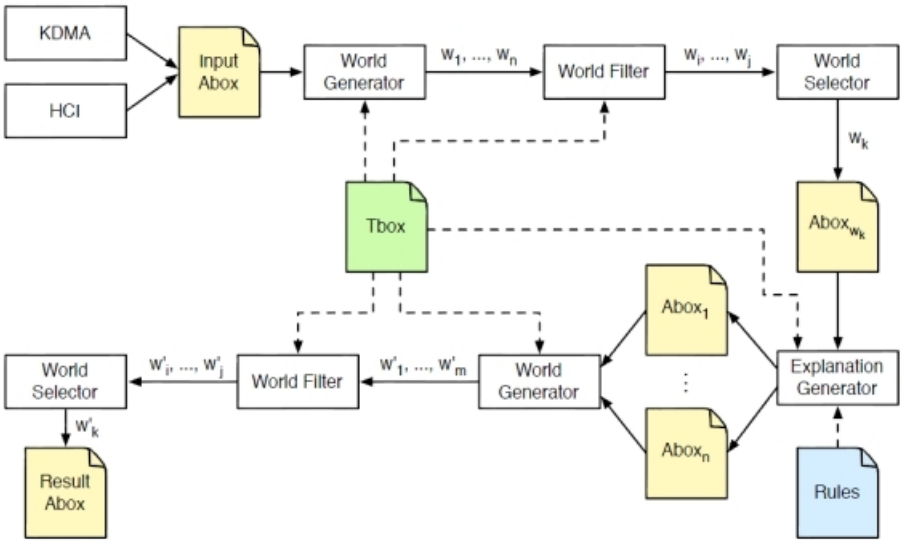


Figure 2.2: Conceptual architecture of the reasoning-based media interpretation engine (from [GMN+09])

possible world is a vector of ground atoms. If one world consists of m ground atoms, the number of possible worlds consequently is 2^m. These generated worlds are the input for the following component.

- **World Filter:** This component takes the output of the previous stage and removes impossible worlds. For this, the subsumption axioms and domain and range restrictions in the TBox are considered for world elimination. The remaining worlds $w_1, w_2, ...w_j$ (also known as possible worlds) are the input to the next component.
- **World Selector:** This component selects the most-probable world among the set of possible worlds. The most-probable world has the highest probability based on Markov logic. The most-probable world w_k is transformed into an intermediate ABox (called ABox_{w_k}).

After the input data passed through the three Markov logic stages, the resulting ABox_{w_k} is used as input for the Explanation Generator. This unit's goal is the derivation of further evidence for the received observations (i. e. the generation of new ("hypothesized") assertions from which the input assertions can be derived). This unit may return multiple output ABoxes (i. e. explanations), all of which are fed back to the three-step process subsequently.

Since the functionality of the Explanation Generator plays a key role in this work, its implementation with the *CASAM Abduction Engine* is described in more detail in the following section. For more information on functionality and implementation of RMI's other components see [GMN+09].

2.4 The CASAM Abduction Engine

The RMI module contains a component called *CASAM Abduction Engine* (CAE) which performs an abductive reasoning task in order to generate so-called explanations for the assertions found in the Input ABox. The goal of computing explanations is to derive additional support for ABox assertions which should be satisfied in all models. The core of the CAE component is the reasoner RacerPro 2.0[4] and its abduction engine.

[4] www.racer-systems.com

Formally speaking, the abduction process is formalized as

$$\Sigma \cup \Delta \models_{\mathcal{R}} \Gamma \tag{2.1}$$

where background knowledge (Σ), rules ($\mathcal{R}$) and observations (Γ) are given and explanations (Δs) are to be computed.

The abduction engine used in the CASAM project is based on the engine already used in the BOEMIE[5] project. Since the functionality of this engine is of utmost importance for this work, this chapter will give a summary of necessary prerequisites and the current implementation's procedures as described in [Kay] and [GMN+09].

The current implementation of CAE adapts this algorithm directly in order to provide first results. However, as described in [GMN+09], the runtime performance of the procedures discussed below do not scale up very well with large rule sets. For future implementations of the Abduction Engine, a modified approach with probabilistic reasoning is planned. Hence, the presented procedures only reflect the working prototype currently in use, rather than the efficiency-optimized engine that is going to be implemented in future versions.

2.4.1 Preliminary definitions and functions

Variable Substitutions

Let $Vars = \{X, Y, ...\}$ be a set of variables, where variables are denoted by capital letters and $Inds = \{i, j, ...\}$ be a set of individuals, where individuals are denoted by lower-case letters. Furthermore, let $\underline{V_1}$, ..., $\underline{V_2}$ be sequences $\langle ... \rangle$ of variables from V and $\underline{z}$ be a sequence of individuals from $Inds$.

A mapping $[X \rightarrow i,\ Y \rightarrow j]$ from variables to individuals is called *variable substitution* and is denoted by σ. σ associates an individual from $Inds$ to each variable from $Vars$. If a substitution is applied to a variable X for which no mapping of the form $[X \rightarrow i]$ exists, then the result is undefined. A variable X for which all required mappings are defined is called *admissible*.

Grounded Conjunctive Queries

Let $\underline{H}, \underline{V_1}, ..., \underline{V_n}$ be sequences of variables and individuals, and let $Q_1, ...,\ Q_n$ denote concept or role names. Then a conjunctive query q has the form

$$q : \{\underline{H} \mid Q_1(\underline{V_1}), ..., Q_n(\underline{V_n})\} \tag{2.2}$$

The concept or role names Q_1, ..., Q_n define a conjunction of so-called *query-atoms*. The variables to the left of the | sign are called *distinguished* variables. They form the *head* of the query q and define the query result. The variables on the right of the | sign form the *body* of the query. Although the head may be of arbitrary length, each variable in $\underline{H}$ must appear in at least one of the $\underline{V_1}$, ..., $\underline{V_n}$. Variables that appear only in the query's body are called *non-distinguished* variables and are existentially quantified.

Within this work only *grounded* conjunctive queries are used, i. e. all (both distinguished and non-distinguished) variables are only bound to explicitly named individuals in the knowledge base.

Finding an answer to a given query with respect to a knowledge base Σ means finding admissible variable substitutions for all variables in the query such that

$$\Sigma \models \{\sigma(Q_1(\underline{V_1}),\ ...,\ \sigma(Q_n(\underline{V_n})\} \tag{2.3}$$

[5] www.boemie.org

A special case is the *boolean* conjunctive query: this is a conjunctive query with a head $\underline{H}$ of length zero. If a variable substitution σ exists such that the relation (2.3) holds, the query result will be *true*, otherwise the answer will be *false*.

Furthermore, the function *transform* is defined that will convert a set of ABox assertions into a boolean conjunctive query.

Rules

A rule r has the following form:

$$r : P(\underline{H}) \leftarrow Q_1(\underline{V_1}),\ ...,\ Q_n(\underline{V_n}) \tag{2.4}$$

where P, Q_1, ..., Q_n denote concept or role names. Rules can be applied to an ABox $\mathcal{A}$ in order to derive new ABox assertions. Thus the function $apply(\Sigma, r, \mathcal{A})$ returns a set of ABox assertions $\{\sigma(P(\underline{H})\})$ if there exists an admissible variable substitution σ such that the answer to the query

$$\{() \mid Q_1(V_1),\ ...,\ Q_n(V_n)\} \tag{2.5}$$

is *true* with respect to $\Sigma \cup \mathcal{A}$. If no such σ can be found, the function *apply* will return the empty set. The formal application of a set of rules $\mathcal{R} = \{r_1,\ ...,\ r_n\}$ is defined as follows:

$$apply(\Sigma, \mathcal{R}, \mathcal{A}) \ = \ \bigcup_{r_i \in \mathcal{R}} apply(\Sigma, r_i, \mathcal{A}) \tag{2.6}$$

When the function $apply(\Sigma, \mathcal{R}, \mathcal{A})$ is called, it is said that the rules in $\mathcal{R}$ are applied to an ABox $\mathcal{A}$ in a *forward-chaining* way.

Scoring of an explanation

Although the use of a set of rules will guarantee a finite number of generated explanations, the size of the space of abducibles may still lead to a tremendous number of possible explanations. In order to reduce the number of results, only "preferred explanations" will be delivered. For the determination of each explanation's preference, the abduction algorithm contains a *scoring function* that evaluates an explanation Δ according to two criteria originally introduced by Thagard [Tha78]: the less hypothesized assertions an explanation contains (*simplicity*) and the more ground assertions (observations) an explanation contains (*consilience*) the higher its preference score. Based on those criteria the preference score S of an explanation can be calculated as follows:

$$S_f(\Sigma, \mathcal{R}, \mathcal{A}, \Delta) := \#\{\alpha \in \mathcal{A} | \Sigma \cup \Delta \models_{\mathcal{R}} \alpha\} \tag{2.7}$$

$$S_h(\Delta) := \#\Delta \tag{2.8}$$

$$S(\Sigma, \mathcal{R}, \mathcal{A}, \Delta) := S_f(\Sigma, \mathcal{R}, \mathcal{A}, \Delta) - S_H(\Delta) \tag{2.9}$$

In other words, the preference score of an explanation is determined by subtracting the number of assertions in an explanation (denoted by S_h) from the number of assertions in the explanation that follow from $\Sigma \cup \Delta$ (denoted by S_f).

In the following algorithm the preference scoring is performed by the function *select-preferred-explanations(Δs)*. It takes a set of explanations Δs as an input and returns only those explanations with a maximal preference score according to the described scoring function.

Additional Functions

For the functionality of the algorithm two more functions are needed. They should not need any comment since they are rather self-explanatory but for the sake of completeness they are defined here as follows:

generate-new-individuals generates a set of new individuals and *ind-from($\mathcal{A}$)* returns the set of individuals from a given ABox $\mathcal{A}$.

2.4.2 The ABox abduction algorithm

The Casam Abduction Engine uses the abduction algorithm provided by RacerPro's function *retrieve-with-explanation*. This function implements the ABox abduction algorithm as a non-standard retrieval service in DLs. As mentioned in the beginning of this section, abduction is formalized as

$$\Sigma \cup \Delta \models_{\mathcal{R}} \Gamma \tag{2.10}$$

where the background knowledge Σ and an observation Γ are given and explanations Δs are to be computed. Σ is a knowledge base that consists of a TBox $\mathcal{T}$ as well as an ABox $\mathcal{A}$, Γ is an ABox assertion of the form $P(\underline{I})$ (with P being a concept or role name from $\mathcal{T}$ and $\underline{I}$ being a set of individuals from $\mathcal{A}$).

Since the abduction algorithm is implemented as a retrieval inference service in DLs, it requires a boolean query as input. Thus the aforementioned function *transform* is applied to the ABox assertion $P(\underline{I})$ in order to obtain a boolean query $\{()|P(\underline{I})\}$.

The algorithm will return a set of hypothesized ABox assertions (called abducibles). Hence, the space of abducibles has to be defined previously which will be done in terms of a set of rules $\mathcal{R}$. According to these rules, the goal of the algorithm is to compute all explanations Δs (i. e. a set of concept and role assertions) with respect to $\mathcal{R}$.

Assertions from the input ABox are divided into two different categories: depending on the application context there may be some assertions that can be taken for granted (i. e. no explanation needs to be derived). Those assertions are called *bonafide assertions*. The other assertions are called *fiat assertions* and require an explanation. In order to decide which assertions are to be explained, the algorithm takes a strategy function Ω as an additional input.

This leads to RacerPro's abduction algorithm shown in Algorithm 1 as it is presented in [Kay]:

Algorithm abduce($\mathcal{T}, \mathcal{A}, \mathcal{R}, \{()|P(\underline{I})\}, \Omega$)

Input: $\mathcal{T}$, $\mathcal{A}$, $\mathcal{R}$, $\{()|P(\underline{I})\}$, Ω
Output: set of explanations: $\Delta\text{s} = \{\Delta_1, \ ..., \ \Delta_n\}$

NewInds $\Leftarrow$ generate-new-individuals
ExtInds $\Leftarrow$ inds-from($\mathcal{A}$)
$\mathcal{BQ} \Leftarrow$ expand($\{()P(\underline{I})\}, \mathcal{R}$)
foreach $\{()|Q_1(\underline{V_1}), \ ..., \ Q_n(\underline{V_n})\} \in \mathcal{BQ}$ **do**
    $\Delta_i = \{Q_1(\sigma(\underline{V_1}, \Omega)), \ ..., \ Q_n(\sigma(\underline{V_n}, \Omega))\}$
    if $\mathcal{T} \cup \mathcal{A} \cup \Delta_i \not\models \bot$ **then**
        $\Delta\text{s} \Leftarrow \Delta\text{s} \cup \{\Delta_i\}$
    end
end
$\Delta\text{s} \Leftarrow$ select-preferred-explanations(Δs)
return Δs

Algorithm 1: The ABox Abduction Algorithm

Additionally, [Kay] gives the following informal description of the steps performed by the abduction algorithm:

1. Generate a set of new individuals called *NewInds*.
2. Get individuals from $\mathcal{A}$ into a set called *ExtInds*.
3. Expand the query $\{()|P(\underline{I})\}$ to obtain the set of sets of boolean conjunctive queries called $\mathcal{BQ}$.
4. Instantiate variables of each boolean conjunctive query in $\mathcal{BQ}$ with individuals from *ExtInds* $\cup$ *NewInds*. The parameter Ω defines the strategy in finding bindings for a variable. Each boolean conjunctive query with instantiated variables is an explanation Δ_i.
5. Check for each explanation Δ_i whether $\mathcal{A} \cup \Delta_i$ is consistent with respect to $\mathcal{T}$. If Δ_i is consistent, then add it to the set of explanations Δ_s.
6. Filter out explanations from Δ_s with respect to some criteria such that Δs contains preferred explanations only.
7. Return the set of preferred explanations (Δs) for the boolean query $\{()|P(\underline{I})\}$

2.4.3 CAE's main procedure

With the knowledge about the ABox abduction algorithm, it is now possible to describe the main procedure of the CASAM Abduction engine currently used for the explanation of ABox assertions. Explaining ABox assertions with respect to a set of rules means the construction of some high-level explanation such that the ABox assertions are entailed. The explanation of an ABox will result in another ABox. The procedure's input will be an ABox Γ containing a set of observations whose assertions are to be explained. Within the algorithm the function *requires fiat* is used to determine whether an assertion is to be explained.

Also, a termination function Ξ is being used. This function provides termination conditions for the procedure and may be used to take resource limitations into account.

According to the previous sections, the further input parameters are a strategy function Ω, the background knowledge consisting of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$, and a set of Rules $\mathcal{R}$.

Putting all of this together leads to the main procedure of the CASAM Abduction Engine as it is currently implemented:

Algorithm CAE($\Omega, \Xi, \mathcal{T}, \mathcal{A}, \mathcal{R}, \{()|P(\underline{I})\}$

Input: $\Omega, \Xi, \mathcal{T}, \mathcal{A}, \mathcal{R}, \{()|P(\underline{I})\}$
Output: $\mathcal{J}$

$\mathcal{J}' := \{\Gamma\}$
repeat
  $\mathcal{J}' := \mathcal{J}$
  $(\mathcal{A}, \alpha) := \Omega(\mathcal{J})$ // $\mathcal{A} \in \mathcal{J}$, $\alpha \in \mathcal{A}$, so that *requires_fiat(*α*)* holds
  $\mathcal{J}' := (\mathcal{J} \backslash \{\mathcal{A}\}) \cup \text{abduce}(\mathcal{T}, \mathcal{A}, \mathcal{R}, \alpha, \Omega)$
until $\Xi(\mathcal{J})$ **or** no $\mathcal{J}$ and α can be selected such that $\mathcal{J}' \neq \mathcal{J}$;
return $\mathcal{J}$

Algorithm 2: CAE's main function

Informally speaking, this procedure loops through all fiat assertions and explains them via the abduction algorithm. After the procedure has finished there will be a set of ABoxes $\mathcal{J}$. Each of the resulting ABoxes will contain one combination of explanations that entails the observation with respect to the given rules. $\mathcal{I}$ will contain one ABox each for *all* combinations of hypothesized assertions Φ that meet the following two criteria[6]:

- The explanation together with the background knowledge entails the observation with respect to the given rules ($\Sigma \cup \Delta \cup \Phi \models_{\mathcal{R}} \Gamma$)
- The explanation received a maximum preference score with respect to simplicity and consilience.

Although both criteria are an evident consequence of the algorithms discussed above it is important to note that a maximum preference score is not necessarily a unique criterion for a single combination of explanations. In fact, the procedure might return a considerable large amount of ABoxes which all passed the function *select-preferred-explanations*.

The remainder of this work is set within the scope of the CASAM Abduction Engine: when the term "Input ABox" is used subsequently, it refers to CAE's input ABox Γ and accordingly, "Result ABox" denotes an element from CAE's result set $\mathcal{J}$.

[6] Of course this is only true if the procedure was able to retrieve all explanations. If a termination condition from Ξ has been met during runtime the results are obviously not complete. For the remainder of this work it is assumed that the procedure does not terminate early.

3 Interpretation of Multimedia Documents

If the RMI component can clearly annotate the multimedia document (i. e. the abduction engine introduced in the previous chapter returns only a single ABox), the explanations are fed back to KDMA and HCI.

However, if RMI is not able to complete the annotation process with its given knowledge (i. e. multiple ABoxes are returned by CAE and thus no unambiguous result can be obtained), it sends certain queries to the HCI component. Those queries are generated during the abduction process and have the goal of augmenting the system's knowledge base by dissolving any ambiguities that may have arisen within the reasoning process.

Based on those queries, the HCI component will prompt the user to answer a set of questions concerning the multimedia document. Any answers a user may provide to these questions are fed back to the RMI component. Based on these answers RMI should be able to eliminate a subset of ABoxes from the abduction engine's result.

In order to illustrate the function of CASAM's Abduction engine described in the previous chapter, this chapter introduces a set of sample scenarios. First, each scenario is described with a set of input assertions (i. e. observations obtained from KDMA and HCI) as well as a set of rules which may be used to explain the given observations. In succession, the process of explanation retrieval is discussed in detail. This shows how RMI generates different high-level interpretations and presents the situations in which additional information has to be requested via the generation of queries.

The first example is rather small and will be introduced with complete rule definitions used by RacerPro. For the sake of simplicity, the later examples will only be presented by an informal description of the necessary steps and rules.

3.1 Structure of multimedia documents

All examples used within this work are multimedia documents built from video and audio clips. The structure of these documents is defined in the *Multimedia Content Ontology* (mco). Figure 3.1 shows the relevant portion of mco's classes used for the structural description of these documents.

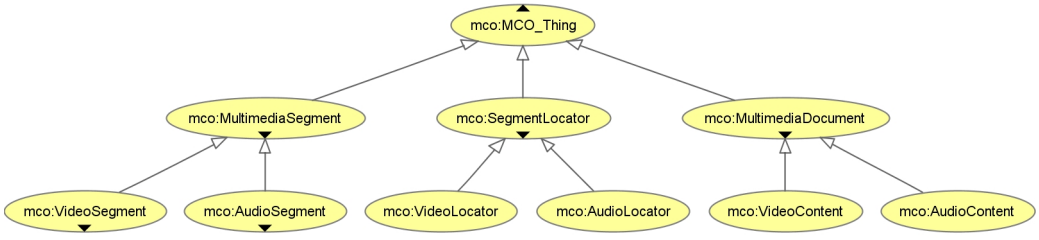


Figure 3.1: mco's classes for the description of audio and video documents

The following list gives a brief description of the classes used within this work as well as important relations that may apply to these classes:

- **mco:MultimediaDocument**
  This is the root class for multimedia documents. This class can be logically decomposed into AudioContent as well as VideoContent. (Actually, this class offers some more content decompositions, but those are not used within this work.)

- **mco:AudioContent**
  This class is the root class for the audio elements within the multimedia document. An object of this class may have one or more audio segments as media decompositions.

- **mco:VideoContent**
  Analogously to the AudioContent class, this is the root class for all video elements.

- **mco:MultimediaSegment**
  Each MultimediaSegment may be a *VideoSegment* or an *AudioSegment*. Each segment represents a part of the clip and has the following properties: *mco:hasSegmentLocator* provides an association between a MultimediaSegment and the corresponding *SegmentLocator*. *mco:depicts* provides an association between a VideoSegment and an individual depicted by this segment. Each segment needs to have exactly one SegmentLocator, but may depict several individuals.

- **mco:SegmentLocator**
  A SegmentLocator (of the subclass *AudioLocator* or *VideoLocator*) is associated with exactly one MultimediaSegment and provides the means to identify the segment's position within the document via the properties *mco:hasStart* as well as *mco:hasEnd*.

- **mco:overlaps**
  This property describes a temporal relation between a VideoLocator and an AudioLocator: it holds if the time interval of a video locator overlaps the time interval of an AudioLocator.

- **mco:meets**
  This property describes a temporal relation between two video locators: a VideoLocator *vl2* meets a VideoLocator *vl1* if the end time of *vl1* has the same value as the start time of *vl2*.

- **mco:hasInterpretation**
  This property applies to multimedia segments that have been interpreted, i. e. some individual has been created based upon this segment. This relation links the segment to the newly created individual.

- **mco:builtFrom**
  This property applies to individuals created throughout the interpretation process and links this new individual to those individuals it is built upon.

3.2 The environmental domain ontology

All examples presented in this work are modeled within the *Environmental Domain Ontology* (edo). This ontology provides a hierarchical collection of classes for all kinds of individuals that might occur in this domain's scope. All of this domain's classes are subclasses of *Thing*. Due to its size, edo's definitions are not presented here as a whole, but a single example should be sufficient to present the idea of edo's hierachical organization: for instance, there's a class *RingingTone* as a subclass of *SoundOfTechnology* as a subclass of *Sound* as a subclass of *PhysicalThing* as a subclass of *EDO_Thing* as a subclass of *owl:Thing* (see Figure 3.2).

Figure 3.2: Example for the hierarchical organization: edo's classes for *RingingTone*

As can be seen in this example, edo's classes are defined rather intuitively. Hence, the classes used for the following scenarios will not be formally defined with respect to edo. Only if more than the obvious properties are needed to comprehend the interpretation process, those additional properties are stated explicitly.

In addition to the class definitions, edo defines relations between certain classes. For instance, it contains a relation *interviews* that links two persons (an interviewer and an interviewee) together.

The subsequent examples follow the naming convention that class names start with capital letters and individual names are created by their corresponding most specific class's name (starting with a lower-case letter for distinction) followed by an index number.

3.3 Sample scenarios

3.3.1 Example 1: construction at the river

Input and Rules

	Segment	depicts
overlaps	vs1	digger1 (ConstructionVehicle)
		brickpile1 (ConstructionResource)
		river1
	as1	constructionNoise1

Table 3.1: relevant properties of the first example's input

In this example, the input ABox contains only two MultimediaSegments: A video segment *vs1* and an overlapping audio segment *al1*. Within the video segment three things were identified: a digger, a pile of bricks and a river. In order to understand the rules used within this example, it is important to know that in the used domain "Digger" is a subclass of "ConstructionVehicle" and "BrickPile" is a subclass of "ConstructionResource". Within the audio segment, the noise of some kind of construction work was identified. The important properties of these assertions are summarized in Table 3.1, the complete ABox Graph created by RacerPorter[1] based upon those assertions is depicted in Figure 3.3.[2]

A few comments are necessary on this input ABox: in this example the input assertions are described as a single ABox that is loaded into RMI. In practice, the situation is somewhat different: KDMA starts analyzing the document and communicates all low-level

[1] RacerPorter is a graphical, interactive interface to RacerPro.

[2] Note that the relations *meets* and *overlaps* are not actually part of the input assertions but they rather have to be inferred by RacerPro via the information about start and end times and a set of corresponding rules. However, since the correlation of different segments is easier to identify through these relations, they are already marked in the input data's summary here. Also, within the reasoning process, those relations are in fact not used on segments but rather on locators. Since each segment is associated with exactly one locator it is possible for the discussion of these examples to augment the relations of locators to their corresponding segments. This prevents the need for a separate listing of each segment's locator.

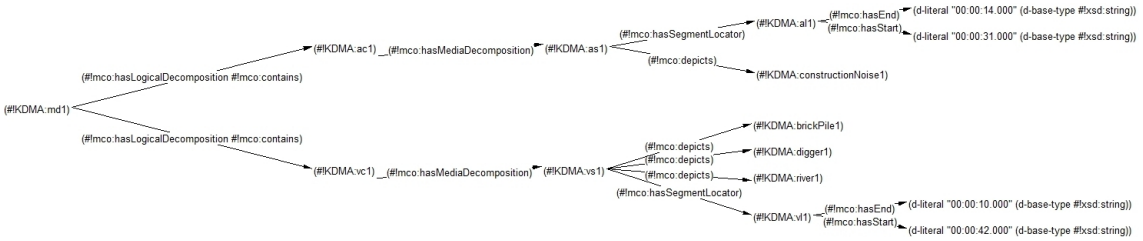


Figure 3.3: The input's ABox Graph for the first example

analysis results to the RMI and HCI components immediately. At the same time, the user may provide some tags to describe the high-level content of the document. These assertions are communicated to KDMA and RMI as well. As a consequence, RMI's input consists of a set of ABoxes with partial information sets about the document, which arrive in different points of time and the knowledge is built up incrementally by merging the different ABoxes into RMI's knowledge base. Also, as described in chapter 2.3, RMI's input assertions are processed by the world generator, filter, and selector stages resulting in an intermediate ABox A_{w_k}. However, since neither the actual source of an assertion nor the exact arrival time at RMI is of any relevance for this work, it is assumed for the sake of simplicity that all assertions in the given sample scenarios are provided as a single ABox which forms CAE's input. Therefore, the term "input assertions" actually refers to A_{w_k} for the remainder of this work.

The presented scenario can be interpreted according to the following rules:

1. <1-> If a video segment shows some construction resource as well as some construction vehicle, there are two possible explanations: the video either shows a construction yard or a construction site. This can be formulated as the following rule:
   inSameSegment(ConstructionVehicle, ConstructionResource)
   ⇒ **ConstructionYard OR ConstructionSite**

2. <2-> If a video segment shows a construction site and an overlapping audio segment contains construction noises, these observations can be explained in two ways: they are either caused by construction or by demolition. These explanations can be formulated as the following rule:
   overlaps(ConstructionSite, ConstructionNoise)
   ⇒ **Construction OR Demolition**

3. <3-> If a construction is detected and the same video segment also depicts a river, this can be explained via either a DamBuilding or a BridgeBuilding:
   inSameSegment(Construction, River)
   ⇒ **DamBuilding OR BridgeBuilding**

Note that the above definitions are a rather informal way to present the rules' essence. For the corresponding implementation of this scenario with RacerPro the rules are divided into two parts: The forward-rules (Figure 3.4) are applied in a forward-chaining way to the ABox in order to identify relations between the given individuals. Based on those relations the backward-rules (Figure 3.5) are used to find explanations for the observations. The retrieval of explanations is implemented by RacerPro's function *retrieve-with-explanation* (see Section 2.4.2 for details).

An examination of RacerPro's rules shows that there is in fact a distinction between input assertions and hypothesized explanations: assertions that are part of the input ABox are object to some *depicts* relationship and thus they are tightly coupled to a certain multimedia segment. For individuals hypothesized during the interpretation, this relationship

```
;;; rules that are applied in a forward-chaining way:

;;; Construction and River in one videosegment => constructionToRiver
(define-rule (?co ?ri |http://www.casam-project.eu/edo.owl#constructionToRiver|)
  (and (?vs |http://www.casam-project.eu/mco.owl#VideoSegment|)
       (?vs ?co |http://www.casam-project.eu/mco.owl#hasInterpretation|)
       (?vs ?ri |http://www.casam-project.eu/mco.owl#depicts|)
       (?co |http://www.casam-project.eu/edo.owl#Construction|)
       (?ri |http://www.casam-project.eu/edo.owl#River|))
        :backward-rule-p nil)

;;; ConstructionResource and ConstructionVehicle in one videosegment => constructionResourceToConstructionVehicle
(define-rule (?cr ?cv |http://www.casam-project.eu/edo.owl#constructionResourceToConstructionVehicle|)
  (and (?vs1 |http://www.casam-project.eu/mco.owl#VideoSegment|)
       (?vs1 ?cr |http://www.casam-project.eu/mco.owl#depicts|)
       (?vs1 ?cv |http://www.casam-project.eu/mco.owl#depicts|)
       (?cr |http://www.casam-project.eu/edo.owl#ConstructionResource|)
       (?cv |http://www.casam-project.eu/edo.owl#ConstructionVehicle|))
  :backward-rule-p nil)

;;; Construction-videosegment overlaps ConstructionNoise-audiosegment => constructionSiteToConstructionNoise
(define-rule (?cs ?cn |http://www.casam-project.eu/edo.owl#constructionSiteToConstructionNoise|)
  (and (?as |http://www.casam-project.eu/mco.owl#AudioSegment|)
                  (?vs |http://www.casam-project.eu/mco.owl#VideoSegment|)
            (?as ?al |http://www.casam-project.eu/mco.owl#hasSegmentLocator|)
      (?vs ?vl |http://www.casam-project.eu/mco.owl#hasSegmentLocator|)
      (?vl ?al |http://www.casam-project.eu/mco.owl#overlaps|)
      (?vs ?cs |http://www.casam-project.eu/mco.owl#hasInterpretation|)
      (?as ?cn |http://www.casam-project.eu/mco.owl#depicts|)
      (?cs |http://www.casam-project.eu/edo.owl#ConstructionSite|)
      (?cn |http://www.casam-project.eu/edo.owl#ConstructionNoise|))
  :backward-rule-p nil)
```

Figure 3.4: RacerPro's forward-rules for the first example

is not provided directly. Instead of using the *depicts* property, a hypothesized individual is object to *hasInterpration* relationssships with all segments involved in the explanation. This difference has to be considered when creating the actual rules; hypothesized individuals have to be treated differently from input individuals. Again, the rules' depiction in this work neglects this difference to simplify their presentation. If hypothesized individuals are created based upon multiple segments, they are considered to have a start value of the first segment's start time and an end value of the last segment's end time.

The rules actually used for the implementation with RacerPro are only listed once for this example to give an idea about their functionality. For the sake of simplicity, the following examples will be discussed only based upon the more informal rule's presentation as shown above.

Interpretation

Based on the provided assertions and rules it is now possible to describe the interpretation process as performed by RMI: Initially the ABox assertions fulfill only the preconditions of the first rule. Hence, this rule is applied to explain *digger1* and *brickPile1*, the other assertions and rules do not have any influence on this interpretation step.[3] Application of rule 1 leads to two different explanations: the video depicts a construction yard or a construction site.[4]

Since the interpretation engine obviously lacks some necessary information at this point,

[3] Note that this explains the aforementioned statement that it is not necessary to consider source or temporal order of the assertions: as long as the preconditions of the first rule are not met, the interpretation process is not able to start. However, as soon as the preconditions are met these assertions can be explained independently of the number or kind of any other possible assertions.

[4] Also, it is always possible that none of the explanations found is right.

```
;;; Rules applied backwards:

;;; Rule 1, Part 1:
;;; ConstructionResource + ConstructionVehicle => ConstructionSite
(define-rule (?cr ?cv |http://www.casam-project.eu/edo.owl#constructionResourceToConstructionVehicle|)
             (and (?cr |http://www.casam-project.eu/edo.owl#ConstructionResource|)
                                 (?cv |http://www.casam-project.eu/edo.owl#ConstructionVehicle|)
                                 (?cs ?cr |http://www.casam-project.eu/mco.owl#builtFrom|)
                        (?cs ?cv |http://www.casam-project.eu/mco.owl#builtFrom|)
                        (?cs |http://www.casam-project.eu/edo.owl#ConstructionSite|))
                        :forward-rule-p nil)

;;; Rule 1, Part 2:
;;; ConstructionResource + ConstructionVehicle => ConstructionYard
(define-rule (?cr ?cv |http://www.casam-project.eu/edo.owl#constructionResourceToConstructionVehicle|)
             (and (?cr |http://www.casam-project.eu/edo.owl#ConstructionResource|)
                                 (?cv |http://www.casam-project.eu/edo.owl#ConstructionVehicle|)
                                 (?cs ?cr |http://www.casam-project.eu/mco.owl#builtFrom|)
                        (?cs ?cv |http://www.casam-project.eu/mco.owl#builtFrom|)
                        (?cs |http://www.casam-project.eu/edo.owl#ConstructionYard|))
                        :forward-rule-p nil)

;;; Rule 2, Part 1:
;;; ConstructionSite + ConstructionNoise => Construction
(define-rule (?cr ?cv |http://www.casam-project.eu/edo.owl#constructionSiteToConstructionNoise|)
             (and (?cr |http://www.casam-project.eu/edo.owl#ConstructionSite|)
                                 (?cv |http://www.casam-project.eu/edo.owl#ConstructionNoise|)
                                 (?cs ?cr |http://www.casam-project.eu/mco.owl#builtFrom|)
                        (?cs ?cv |http://www.casam-project.eu/mco.owl#builtFrom|)
                        (?cs |http://www.casam-project.eu/edo.owl#Construction|))
                        :forward-rule-p nil)

;;; Rule 2, Part 2:
;;; ConstructionSite + ConstructionNoise => Destruction
(define-rule (?cr ?cv |http://www.casam-project.eu/edo.owl#constructionSiteToConstructionNoise|)
             (and (?cr |http://www.casam-project.eu/edo.owl#ConstructionSite|)
                                 (?cv |http://www.casam-project.eu/edo.owl#ConstructionNoise|)
                                 (?cs ?cr |http://www.casam-project.eu/mco.owl#builtFrom|)
                        (?cs ?cv |http://www.casam-project.eu/mco.owl#builtFrom|)
                        (?cs |http://www.casam-project.eu/edo.owl#Destruction|))
                        :forward-rule-p nil)
;;; Rule 3, Part 1:
;;; Construction + River => DamBuilding
(define-rule (?cr ?cv |http://www.casam-project.eu/edo.owl#constructionToRiver|)
             (and (?cr |http://www.casam-project.eu/edo.owl#Construction|)
                                 (?cv |http://www.casam-project.eu/edo.owl#River|)
                                 (?cs ?cr |http://www.casam-project.eu/mco.owl#builtFrom|)
                        (?cs ?cv |http://www.casam-project.eu/mco.owl#builtFrom|)
                        (?cs |http://www.casam-project.eu/edo.owl#DamBuilding|))
                        :forward-rule-p nil)

;;; Rule 3, Part 2:
;;; Construction + River => BridgeBuilding
(define-rule (?cr ?cv |http://www.casam-project.eu/edo.owl#constructionToRiver|)
             (and (?cr |http://www.casam-project.eu/edo.owl#Construction|)
                                 (?cv |http://www.casam-project.eu/edo.owl#River|)
                                 (?cs ?cr |http://www.casam-project.eu/mco.owl#builtFrom|)
                        (?cs ?cv |http://www.casam-project.eu/mco.owl#builtFrom|)
                        (?cs |http://www.casam-project.eu/edo.owl#BridgeBuilding|))
                        :forward-rule-p nil)
```

Figure 3.5: RacerPro's backward-rules for the first example

it will send a query to HCI in order to decide whether the scene depicts a construction yard or a construction site (or neither). Details of the queries are described in the following chapter.

Although it is not possible to compute an unambiguous explanation at this point, due to efficiency considerations it would not be wise to pause the interpretation process until the necessary information is obtained. Instead of waiting for the query's answer, RMI will consider both possibilities and proceed with the interpretation on different paths. Hence, the current ABox is cloned and one of the possible explanations is added to each of the ABoxes subsequently. As a result, RMI maintains two ABoxes in parallel: ABox1 contains the interpretation *ConstructionYard* while ABox2 contains *ConstructionSite*.[5]

Note that the interpretation step does not always have to return two different explanations. Depending on the given assertions and rules the number of generated explanations may vary. Only if more than one explanation can be found it is necessary to clone the ABox. Obviously, the number of ABox clones is determined by the number of possible explanations.

In the next step RMI tries to continue the explanation of each ABox: The Assertions of ABox1 do not meet the preconditions of any unprocessed rule. Thus, no further interpretation is possible and this interpretation path ends. Through the additional assertion *ConstructionSite* in ABox2 and the initial assertion *constructionNoise1*, the preconditions of rule 2 are met and therefore this rule is applied. Again, this leads to two possible explanations. Thus, a new query is generated, the ABox is cloned again, and the explanation *Demolition* is added to the resulting ABox3, while *Construction* is added to ABox4.

RMI continues to search for interpretations for each of the ABoxes: ABox3 does not meet the precondition for any other rule and thus marks another end of an interpretation path. The additional assertion *Construction* together with the assertion *river1* in ABox4 fulfills the preconditions for rule 3 and thereby this ABox can be explained further. Application of this rule leads to two different explanations yet again. Hence, another query is generated, the ABox is cloned again, and the explanation *DamBuilding* is added to ABox5 and *Bridgebuilding* is added to ABox6.

At this point none of the ABoxes can be interpreted any further (i. e. all ABoxes form leaves in the interpretation tree) and thus the interpretation process for this scenario is finished.

A visualization of the described interpretation process is shown in Figure 3.6: ABoxes are depicted as white ellipses, final ABoxes (i. e. ABoxes which cannot be explained any further) are marked with an underlying shade additionaly. Interpretation steps are represented as green diamonds and their captions denote the observations which are explained in each particular step. The corresponding rules for each explanation step are specified within the blue rectangles, any queries that may arise during the interpretation are denoted as red arrows. The resulting hypotheses of an explanation (i. e. the additional assertions that have to be added to subsequent ABoxes) are marked on the arrows originating from the explanation step, respectively.

As depicted, the interpretations (and the corresponding queries) are categorized into different levels: first-level interpretations are built directly from the input observations, second-level interpretations are built from first-level interpretations and so on. Note that it is not necessary for a high-level interpretation to be built completely from interpretations of the next-lower lever, but it can also include assertions from lower levels. The actual level of each interpretation is one level above the highest level of the assertions it is built from. For instance, the interpretation *DamBuilding* in the above example is of level three because

[5] Actually, the individuals created by RacerPro during the interpretation process are named *IND-1*, *IND-2*, and so on. However, in order to achieve an easier comprehension of the examples, the interpretation results are presented via their corresponding class names rather than their actual (quite expressionless) individual names.

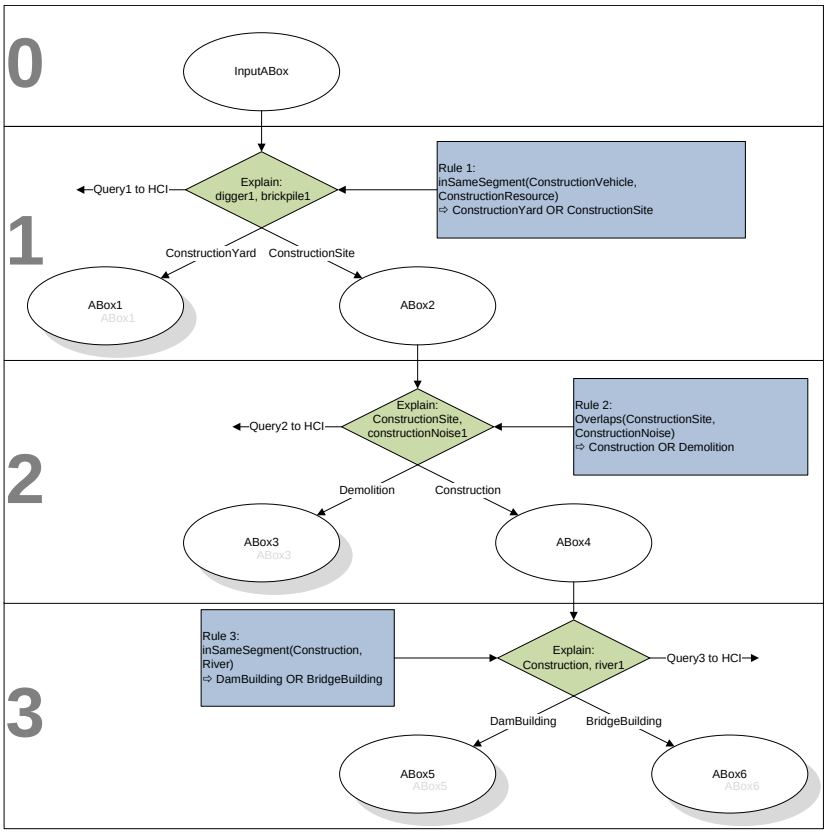


Figure 3.6: Tree view of the first example's interpretation process

it involves the second-level interpretation *Construction* as well as the input (or "level zero") assertion *river1*. Formally speaking, the level of an interpretation is determined by its longest chain of *builtFrom* relations.

3.3.2 Example 2: smoke in the sky

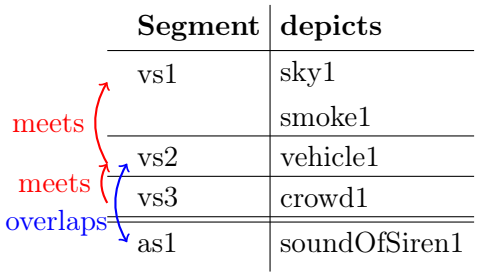
Input and Rules

This example is a bit more complex than the previous one. The input consists of three video segments and one audio segment. In the first video segment *vs1* some smoke and the sky is detected. The next segment *vs2* depicts some kind of vehicle while the last segment *vs3* depicts a crowd. In the audiosegment the sound of a siren could be detected. While it is not necessary to know the exact timing of each segment it is important to note that *vs2* meets *vs1* and *vs3* meets *vs2*. Also, the audio segment is supposed to overlap with the second video segment. The input's relevant properties are summarized in Table 3.2, the corresponding ABox graph is shown in Figure 3.7.

For the interpretation of this scenario, the following set of rules is being used:

1. If a video segment depicts the sky, it can be concluded that some kind of outdoor location is shown.
   Sky
   ⇒ **OutdoorLocation**

2. If a videosegment shows an outdoor location and smoke is depicted within this segment, there might be two explanations for the smoke's source: there is either a fire or a factory.



Segment	depicts
vs1	sky1
	smoke1
vs2	vehicle1
vs3	crowd1
as1	soundOfSiren1

Table 3.2: relevant properties of the second example's input

inSameSegment(OutdoorLocation, Smoke)
⇒ **Factory OR Fire**

3. If a video segment shows a vehicle and at the same time an audio segment depicts a siren sound, it can be reasoned that the vehicle is either a police car or a fire engine.
overlaps(Vehicle, SoundOfSiren)
⇒ **PoliceCar OR FireEngine**

4. If one video segment depicts a crowd and the following segment depicts a police car (i. e. the police car meets the crowd), there are two explanations for this scenario: this event either depicts a demonstration observed by the police, or it is a conference (for example with a prominent person attenting or with an explosive topic and therefore requiring attendance of the police).
meets(PoliceCar, Crowd)
⇒ **Conference OR Demonstration**

5. If the crowd is met by a fire engine instead of a police car, it can be assumed that some kind of emergency happened that required the presence of a fire engine and that the crowd is made up by curious onlookers.
meets(FireEngine, Crowd)
⇒ **EmergencyEvent**

6. If a video segment is interpreted as a fire and this segment is met by some kind of emergency event, it is likely that the fire engine is present to extinguish the fire.
meets(EmergencyEvent, Fire)
⇒ **FireExtinguishEvent**

7. If a video segment is interpreted as a factory and this segment is met by a demonstration, it can be assumed that there is a demonstration against the factory.
meets(Demonstration, Factory)
⇒ **DemonstrationAgainstFactory**

Note that the rules presented here impose a rather tight coupling of the assertions' temporal relations. In order to capture the meaning of this scenario such a tight coupling is not necessary. In fact, the rules actually implemented might be triggered by several similar relations. Take a look at rule 4 for an instance: the order of appearance of the police car and the crowd may be switched around without changing the meaning of the scene. Also, it could happen that both observations are depicted within the same segment. Obviously, a reliable interpretation algorithm should come to the same conclusions in this case independently of the exact order. However, in order to keep the rules clear, only preconditions occuring in this example are presented, while the expedient rule set used by RacerPro actually contains several supplementary rules in order to capture situations with obersavations of equivalent substance.

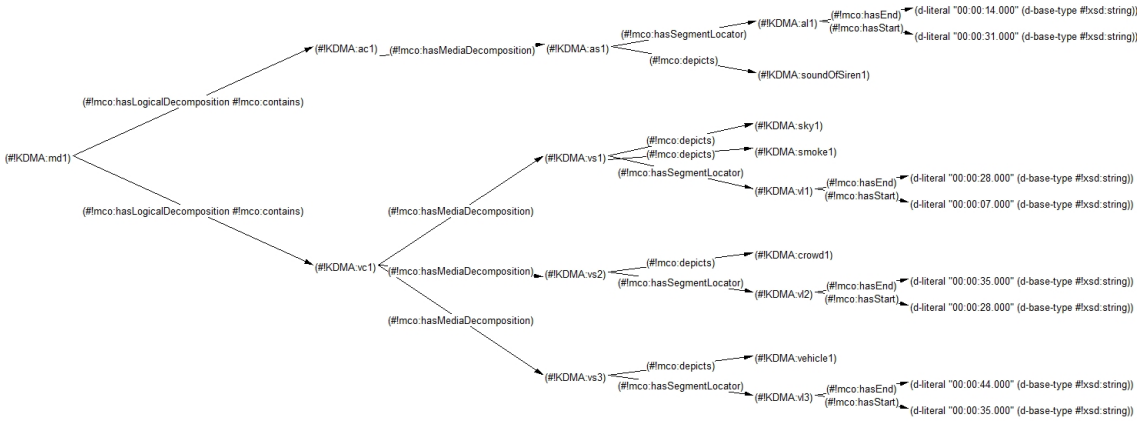


Figure 3.7: The input's ABox Graph for the second example

Interpretation

Now it is possible to describe the interpretation process of this scenario as depicted in Figure 3.8: since the first video segment depicts the sky, it can be concluded by rule 1 that an outdoor location is shown. Since no other explanation is possible, this assertion is directly added to the current ABox.[6] Also, the assertions *vehicle1* and *sirensound1* meet the preconditions of rule 3. Since this rules lead to two possible explanations, the current ABox is cloned, a query is send to HCI and the assertion *PoliceCar* is added to ABox2 and *FireEngine* is added to ABox3.

In the next step rule 2 is examined: both of the existing ABoxes meet the rule's preconditions (*OutdoorLocation* and *smoke1*), so the same interpretation step has to be performed on both ABoxes. Since the interpretation leads to two possible explanations, each of the ABoxes is cloned and a corresponding query is send for *each* cloning process, respectively. As a result, there are now four different ABoxes: ABox4 gets the assertion *Factory* and ABox5 gets the assertion *Fire*, both of them contain the previous assertion *PoliceCar*. In the same way, the explanations are added to ABox6 and ABox7 respectively, but they contain the assertion *FireEngine* instead of *PoliceCar*.

In the following, rule 4 is examined: two of RMI's currently maintained ABoxes (ABox4 and ABox5) fulfill the preconditions of this rule (*PoliceCar* and *crowd1*). As a result, both of these ABoxes are cloned, a corresponding query is send to HCI for each cloning process, and the rule's possible explanation *Conference* is applied to ABox8 and ABox10 and *Demonstration* is applied to the ABox10 and ABox11 respectively.

In a similar way rule 5 is applied to ABox6 and ABox7: both of them meet the rule's precondition (*FireEngine* and *crowd1*). This leads to the single explanation *EmergencyEvent*, which is applied to both ABoxes resulting in ABox12 and ABox13.

At this point, all second level interpretations have been performed and RMI currently maintains six different ABoxes. Only two of them offer an option to continue the interpretation process:

ABox9 contains the assertions *Demonstration* and *Factory* in adjacent video segments and thus the preconditions for rule 7 are met. Hence, this situation leads to the third level interpretation *DemonstrationAgainstFactory*.

[6] Since only a single explanation could be retrieved, no ambiguities arise during this step and RMI is able to continue the interpretation process on a unique path. Hence, no query is *necessary* at this point to proceed towards a unique solution. However, the occurrence of only a single explanation to a certain observation does not necessarily guarantee the correctness of this explanation. Since the continuation of the interpretation process will rely on this explanation, it might still be *desirable* to verify this result via a query to HCI. The question whether this situation should produce a query or not will be discussed in the next chapter.

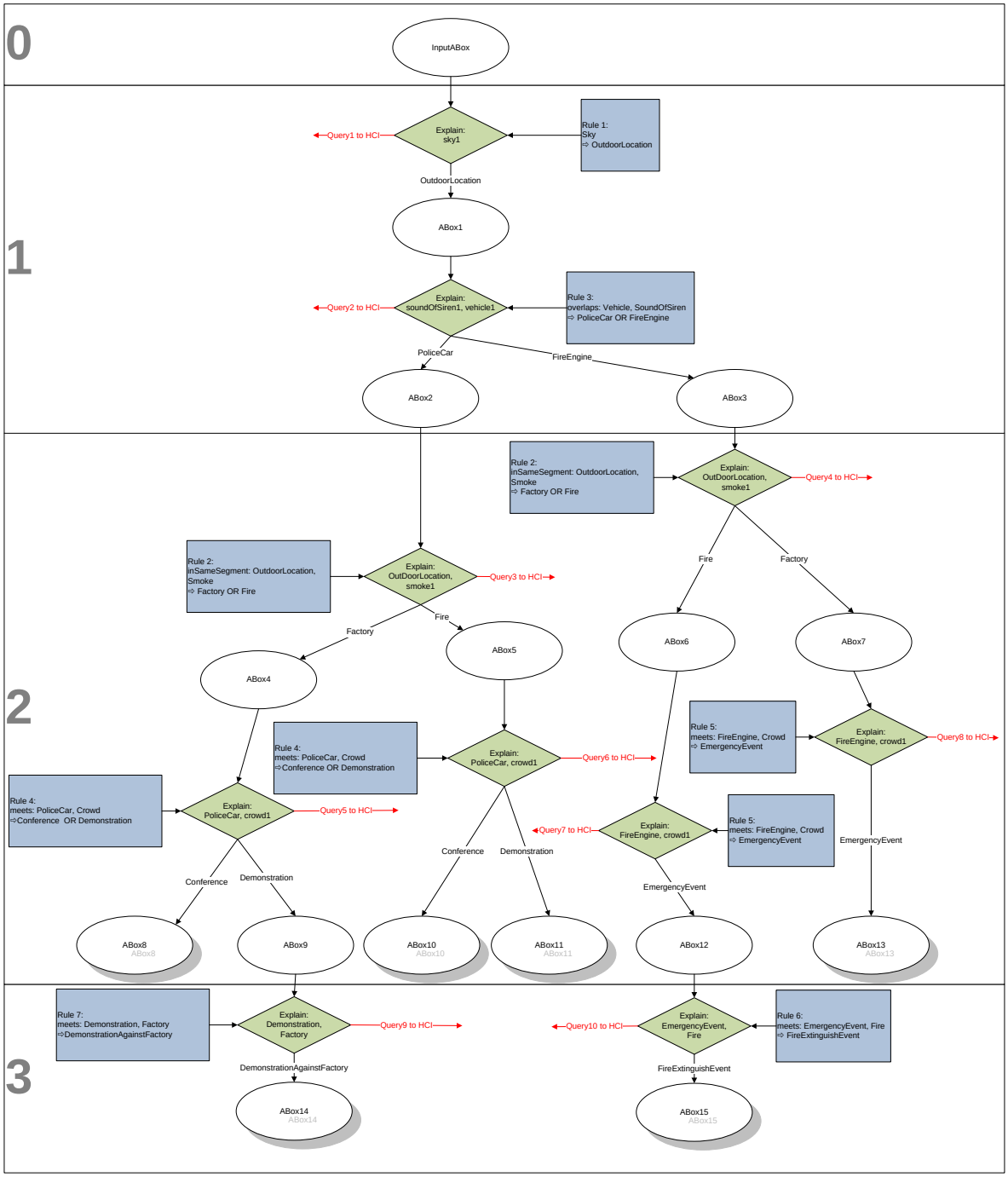


Figure 3.8: Tree view of the second example's interpretation process

ABox12 fulfills the criteria for rule 6 (*EmergencyEvent* and *Fire*), which consequently leads to the interpretation *FireExtinguishEvent*.

Since at this point nothing more can be inferred from the given assertions and rules, RMI's interpretation process terminates here.

Optimization of the Interpretation Process

The interpretation process discussed above is the result of CAE's main procedure as described in Section 2.4.3. While this algorithm worked flawlessly for the first example, this example exposes some drawbacks of the current implementation: *every* interpretation that produces multiple results yields to a branch in the interpretation path and an independent interpretation is performed on each branch subsequently. This is not a problem as long as only one of the available rules is applicable to each ABox (as it was the case in example 1). However, if more than one rule is applicable to the current ABox, one of the rules has to be applied first and thus a branch is created. In the next step, another rule that has been applicable before is applied to all of the previous interpretation's results. Hence, the same interpretation is performed multiple times and thus computing resources are wasted and (probably even worse) multiple queries with the same content will be sent to the user.

For instance, take a look at the police car branch in the previous example: for ABox1 rule 3 is applicable as well as rule 2. Since rule 3 has been applied first, the interpretation based on rule 2 has to be carried out twice, although it could have been applied to ABox1 also.[7]

A possible solution to this problem could be a different way of handling the interpretation branching: if at some point more than one rule is applicable to the current ABox, RMI would branch the interpretation into different paths *before* the first rule is applied. Each possible rule is applied subsequently and the generated explanations lead to ABox-clones as they have done before. This way, RMI already has knowledge about further additions to the currently maintained ABoxes without actually adding this knowledge yet. Instead, the ABoxes are interpreted as far as possible with the current rules and assertions. Only if nothing more can be inferred from the current set of assertions, the assertions from the other branch are added. Although this method of holding back assertions might not appear desirable at first glance, since some rules could be fired sooner with the missing assertions, a closer look at the process shows that the number of required interpretation steps could possibly be reduced significantly.

The modified process for the second example is shown in 3.9: instead of applying rule 3 to ABox1 and subsequently applying rule 2 to the resulting ABoxes of rule 3, both rules are actually directly applied to ABox1. As a result, two different major branches evolve whose hypotheses are not mutually exclusive (i. e. explanations from one branch are also possible explanations in the other branch). Without actually adding the resulting explanations to each branch, both of them are separately interpreted as far as possible, while at the same time the knowledge about the existence of additional hypotheses is maintained. Once all possible interpretations have been carried out, the different branches are merged together into all possible combinations of explanations. This merging step (depicted by the colored arrows) substitutes the "clone-and-add-explanations-step" that has been skipped for rule 3. Subsequently, the resulting ABoxes after the merging step can be subject to further interpretations.

Note that the results of this method are the same as before, but three out of ten interpretation steps (and thereby three corresponding queries to the user) were saved.

[7] There is no specification as to which rule has to be applied first. Of course, rule 2 could have been applied to ABox1 first. However, this would not have any effect on the problem, since the algorithm's functionality would apply rule 3 twice subsequently.

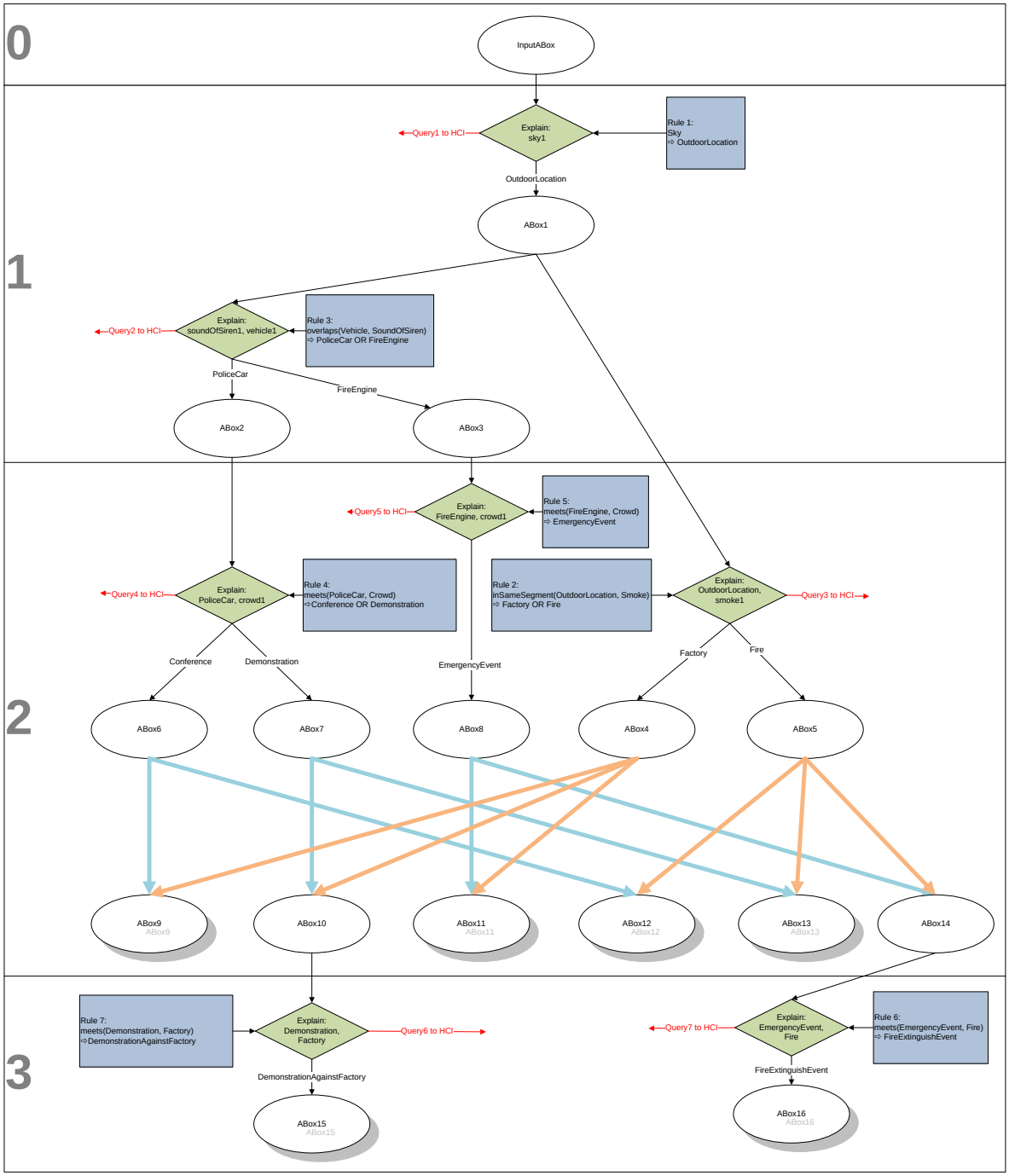


Figure 3.9: Tree view of the second example's optimized interpretation process

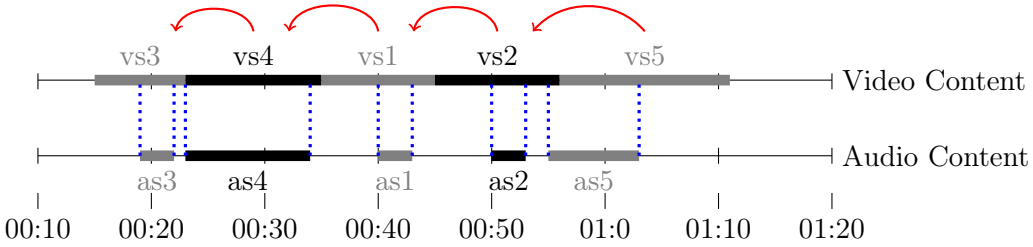


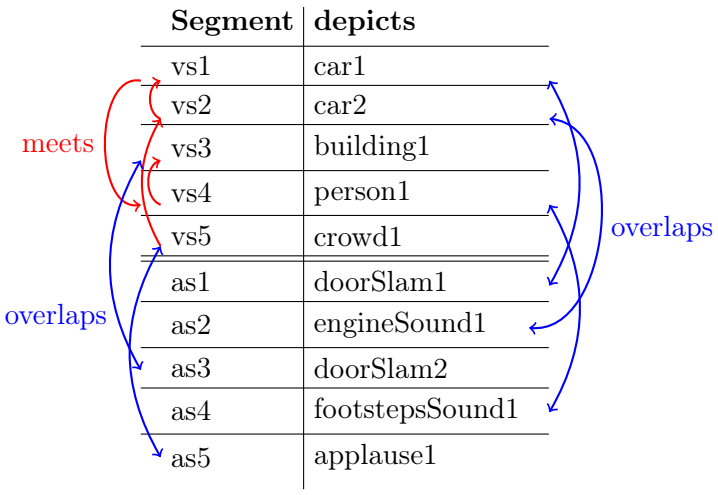
Figure 3.10: Timeline for example 3

3.3.3 Example 3: car-related events

Input and Rules

The last example presented here is a film sequence that consists of five audio segments and five video segments. All observations come as pairs of overlapping audio and video segments: the first segment shows a car overlapping with the noise of a door slam. The second pair shows a car also, this time overlapping with the sound of an engine. In the third pair, a building is depicted that overlaps with the sound of another door slam. The fourth pair shows a person and the noise of foot steps is present. In the last pair the video segment shows a crowd while the audio segment depicts applause.

Also, there are important temporal relations between the video segments: *vs3* is the first segment in this document, directly followed by *vs4*. *vs4* is followed by *vs1*, *vs1* is followed by *vs2*, and *vs2* is followed by *vs4*. These properties are summarized in the following table. Since the timing is rather important in this example, the information from Table 3.3 is additionally visualized on a time line in Figure 3.10. The corresponding graph is depicted in Figure 3.11.



Segment	depicts
vs1	car1
vs2	car2
vs3	building1
vs4	person1
vs5	crowd1
as1	doorSlam1
as2	engineSound1
as3	doorSlam2
as4	footstepsSound1
as5	applause1

Table 3.3: relevant properties of the third example's input

This ABox is going to be interpreted according to the following rules:

1. If a car depicted in a video segment overlaps with the noise of a slammed door, this event may be either a car entry or a car exit.
   overlaps(Car, DoorSlam)
   ⇒ **CarEntry OR CarExit**

2. If a car overlaps with some kind of engine sound, it can be assumed that it is a car

ride.
overlaps(Car, EngineSound)
⇒ **CarRide**

3. The overlapping of a building with the noise of a slammed door can be interpreted either as a building entry or a building exit.
overlaps(Building, DoorSlam)
⇒ **BuildingEntry OR BuildingExit**

4. If the video shows a person and the audio depicts the sound of footsteps, its probably a person walking.
overlaps(Person, FootstepsSound)
⇒ **Walk**

5. A crowd in the video segment overlapping with the sound of applause in the audio segment could be explained as a cheering crowd.
overlaps(Crowd, Applause)
⇒ **CheeringCrowd**

6. If a car entry is followed by a car ride, this can be explained with the event of somebody getting into a car and driving away.
meets(CarRide, CarEntry)
⇒ **LeavingEvent**

7. If a car ride is followed by a car exit, it could be a car arriving and the driver leaving the car.
meets(CarExit, CarRide)
⇒ **ArrivingEvent**

8. The two rules defined above seem to be the obvious observations for either an arrival or an departure. However, it might be possible that the person entering or exiting the car is not identical to the driver. In this case, two different events are thinkable (e. g. imagine the car being a cab). If a car exit is followed by a car ride, this could be explained by somebody being dropped off and the car driving on afterwards.
meets(CarRide, CarExit)
⇒ **DriveOnEvent**

9. Also, it might be possible that somebody is picked up by car, in this case the car entry would be preceded by a car ride.
meets(CarEntry, CarRide)
⇒ **PickUpEvent**

10. If a walk is followed by a building entry, it can be interpreted as a person entering a building.
meets(BuildingEntry, Walk)
⇒ **PersonEntersBuilding**

11. In the same way as shown in the previous rule, a building exit followed by a walk can be interpreted as a person leaving a building.
meets(Walk, BuildingExit)
⇒ **PersonLeavesBuilding**

12. The Explanations from the previous rules can be used to obtain interpretations on a higher level: If a person leaves a building and in the following a leaving event takes place, it can be assumed that a person left a building, got into a car, and drove away.

meets(LeavingEvent, PersonLeavesBuilding)
⇒ PersonLeavesBuildingAndDrivesAway

13. Since according to the previous rules a person might leave via a pick up event as well, the same event as before might be inferred by a different input:
meets(PickUpEvent, PersonLeavesBuilding)
⇒ PersonLeavesBuildingAndDrivesAway

14. If an arriving event or a drive on event is followed by a person entering a building this can be interpreted similarly to the previous scenarios leading to two more rules:
meets(PersonEntersBuilding, ArrivingEvent)
⇒ PersonArrivesAndEntersBuilding

15. **meets(PersonEntersBuilding, DriveOnEvent)**
⇒ PersonArrivesAndEntersBuilding

16. If somebody arrives by car (either depicted via an arriving event or a drive on event) and this event is followed by a cheering crowd, it can be explained by the arrival of some prominent person that is greeted by the crowd. This leads to two more rules:
meets(CheeringCrowd, ArrivingEvent)
⇒ ProminentPersonArrives

17. **meets(CheeringCrowd, DriveOnEvent)**
⇒ ProminentPersonArrives

Before proceeding to this example's interpretation, a few comments on the described scenario are necessary. Note that the examples presented previously were fairly independent of their exact temporal relations. Opposed to those examples, the current scenario's rules heavily rely on exact timing. For instance, the observations of a building overlapping with a door slam followed by a person walking are considered. According to the presented rules, this could be interpreted as a person *leaving* a building. However, if the same observations would occur in a different order (i. e. a person walking, followed by a building overlapping with a door slam) the previous interpretation would not hold any longer, but instead this observation could be interpreted as the event of a person *entering* a building instead. Obviously those two explanations have quite a different meaning and could lead to different higher-level interpretations that are highly dissimilar to each other (as shown later on in the discussion of possible interpretations).

If a switched order of events would lead to a different interpretation, the corresponding rules for both situations are presented. This shows how the interpretation of given observations might lead into different directions solely depending on the observations' order. In order to maintain an easy overview of the rules necessary for the described input's interpretation, rules that do not apply to this example are listed in gray.

Interpretation

In order to keep the presentation of the interpretation's process as simple as possible, this example is processed only according to the optimization suggested in the previous example. See Figure 3.12 for a visualization of the process.

In this example, there are quite a few explanations that can be directly inferred from the input's data: from *car2* and *engineSound1* rule 2 can lead to a *CarRide*, *person1* and *footstepsSound1* produce a *Walk* via rule 4, and *crowd1* and *applause1* lead to a *CheeringCrowd* through rule 5. All of these rules only lead to a single explanation each and thus can be applied conveniently in an arbitrary order to the input ABox.

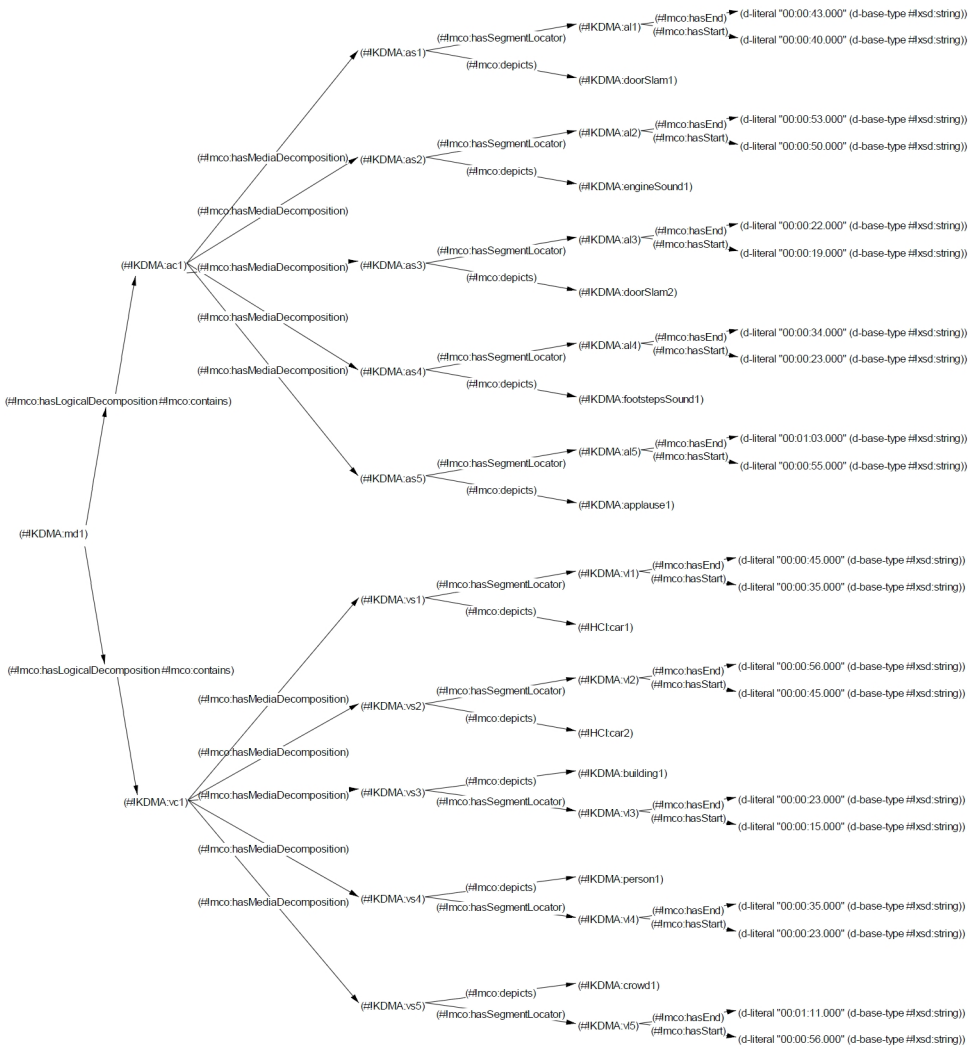


Figure 3.11: The input's ABox Graph for the third example

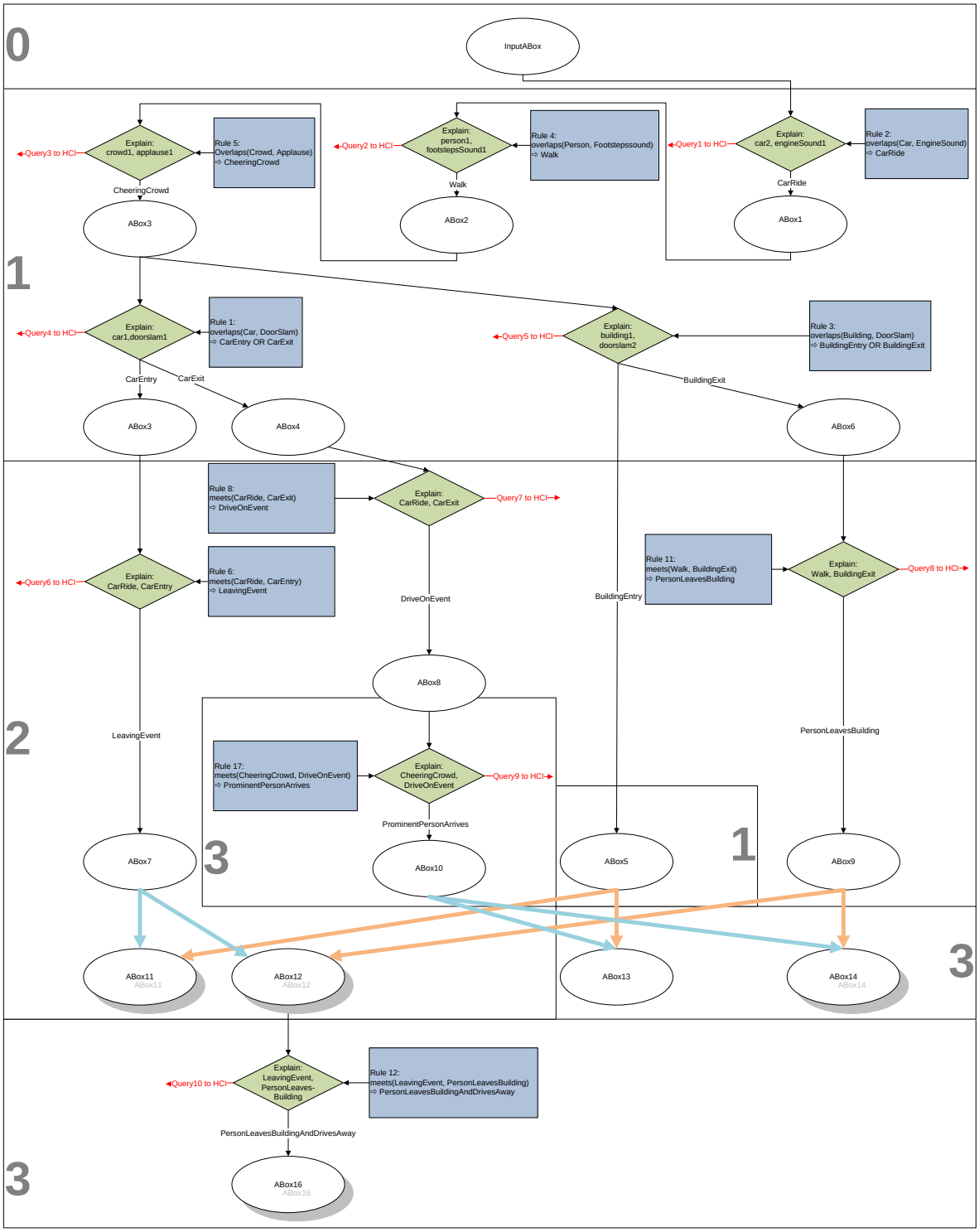


Figure 3.12: Tree view of the third example's interpretation process

Also, the input assertions fulfill the preconditions of two further rules. Since both of those rules yield to more than one explanation, the interpretation branches into two different paths as described in Section 3.3.2.

From *car1* and the overlapping *doorSlam1* it can be concluded via rule 1 that this event depicts either a *CarEntry* or a *CarExit* (stored in ABox3 and ABox4, respectively). In the same way *building1* and *doorSlam2* combined with rule 3 lead to either a *BuildingEntry* or a *BuildingExit* (ABox5 and ABox6).

Continuing on the path containing the car entry (in ABox3), rule 6 is applicable now to the sequence of *CarEntry* and *CarRide* and thus this sequence can be explained as a *LeavingEvent*. The resulting ABox7 cannot be further explained by the rules available.

The situation on the car exit path is quite similar, the sequence of the *CarEntry* and *CarRide* can be explained through rule 8 as a *DriveOnEvent* and this is stored in ABox8.

Now the interpretation goes back up one level and switch to the other path where rule 3 had been applied. For ABox5 (containing the *BuildingEntry*) no more rules are applicable so this path can not be continued for the moment.

For the next ABox (ABox6 containing the *BuildingExit*) rule 11 is now applicable because the *BuildingExit* is met by the *Walk*. Thus, application of rule 11 leads to *PersonLeavesBuilding*. For the resulting ABox9 no more rules apply either, so the whole path containing the building-related events can be disregarded for now.

Switching back to the path with the car-related events, only ABox8 can be further interpreted: since this ABox contains *CheeringCrowd* as well as *DriveOnEvent*, rule 17 can be used for its interpretation and the event *ProminentPersonArrives* is obtained.

Now all currently maintained ABoxes are interpreted as far as possible. Hence, the two different paths are merged together, leading to the following four ABoxes: ABox11 contains *LeavingEvent* and *BuildingEntry*, ABox12 *LeavingEvent* and *PersonLeavesBuilding* (and thereby also *BuildingExit*), ABox13 contains *ProminentPersonArrives* (and thereby also *DriveOnEvent*), and finally ABox 14 contains *ProminentPersonArrives* and *PersonLeavesBuilding*.

Of the resulting set of ABoxes, only ABox12 is applicable to further rules: from the sequence of *PersonLeavesBuilding* and *LeavingEvent* it can be inferred via rule 12 that this sequence depicts the event *PersonLeavesBuildingAndDrivesAway*.

Now all ABoxes are interpreted exhaustively with respect to the available rule-set. Hence, RMI terminates the interpretation process here.

4 Query-generation

As discussed in the previous chapter, RMI creates a query for each explanation step that leads to multiple explanations. This chapter will discuss details about those queries. First of all, it should be noted that a query is created automatically within RacerPro for each interpretation step (see section 2.4.3) that returns multiple explanations. In order to describe the initial situation, this chapter starts with the discussion of the information about an interpretation step that can be directly obtained by RacerPro.

4.1 An interpretation step in RacerPro

To describe the information provided by RacerPro a sample interpretation step is examined here in more detail. Consider the first interpretation step from the "Construction at the River"-Example: this step searches for an explanation for the occurrence of a brick pile and a digger within the same video segment.

4.1.1 Function call for explanation retrieval

The corresponding function call sent to RacerPro is depicted in the following figure. It instructs RacerPro to start the abduction algorithm as described in Section 2.4.2 in order to retrieve explanations for the specified observations.

```
(retrieve-with-explanation nil
       (#!KDMA:brickPile1 #!KDMA:digger1
        #!edo:constructionResourceToConstructionVehicle)
       :final-consistency-checking-p t :only-best-p t :show-score-p t
       :order-by :new-paper-fn :equi-order-by :prefer-new-inds)
```

Figure 4.1: RacerPro's function call to obtain explanations for the co-occurrence of the digger and brick pile in the same video segment

4.1.2 RacerPro's reply

After completing the abduction algorithm, RacerPro replies with a description of possible explanations that would entail the given observation. RacerPro's reply to the aforementioned explanation step is listed in Figure 4.2.

This answer has the following meaning: each explanation found by RacerPro starts with the keyword `(:tuple)`, ergo two explanations were found: lines 2-21 describe the first explanation, lines 22-41 describe the second one. To explain the relevant parameters of this answer, the first example is examined in more detail. The most important part of this answer is found in lines 3-7: line 3 states that one new individual has been created and named *IND-3*. The following lines name the assertions that had to be hypothesized in order to create the new individual, namely that *IND-3* is of the class *ConstructionSite* (line 6) and that is has been built from the brick pile and the digger (lines 5 and 7, respectively).

The following lines provide details about the explanation's scoring (see Section 2.4.1 for details on the scoring function): two assertions are entailed (lines 12-15) and three assertions are hypothesized (lines 17-21). Hence, this explanation's score is -1.

```
(t
  (((:tuple)
    (:new-inds IND-3)
    (:hypothesized-assertions
     (related IND-3 #!KDMA:brickPile1 #!mco:builtFrom)
     (instance IND-3 #!edo:ConstructionSite)
     (related IND-3 #!KDMA:digger1 #!mco:builtFrom))
    (:score
     -1
     :new-paper-fn
     ((:old-inds 2 #!KDMA:brickPile1 #!KDMA:digger1)
      (:entailed-assertions
       2
       (:instance #!KDMA:digger1 #!edo:ConstructionVehicle)
       (:instance #!KDMA:brickPile1 #!edo:ConstructionResource))
      (:new-inds 1 IND-3)
      (:hypothesized-assertions
       3
       (related IND-3 #!KDMA:brickPile1 #!mco:builtFrom)
       (instance IND-3 #!edo:ConstructionSite)
       (related IND-3 #!KDMA:digger1 #!mco:builtFrom)))))
   ((:tuple)
    (:new-inds IND-2)
    (:hypothesized-assertions
     (related IND-2 #!KDMA:brickPile1 #!mco:builtFrom)
     (instance IND-2 #!edo:ConstructionYard)
     (related IND-2 #!KDMA:digger1 #!mco:builtFrom))
    (:score
     -1
     :new-paper-fn
     ((:old-inds 2 #!KDMA:brickPile1 #!KDMA:digger1)
      (:entailed-assertions
       2
       (:instance #!KDMA:digger1 #!edo:ConstructionVehicle)
       (:instance #!KDMA:brickPile1 #!edo:ConstructionResource))
      (:new-inds 1 IND-2)
      (:hypothesized-assertions
       3
       (related IND-2 #!KDMA:brickPile1 #!mco:builtFrom)
       (instance IND-2 #!edo:ConstructionYard)
       (related IND-2 #!KDMA:digger1 #!mco:builtFrom)))))))
```

Figure 4.2: RacerPro's reply to the explanation-retrieval

The second explanation is very similar to the first one, except that the new individual created here is *IND-2* of the class *ConstructionYard*.

4.1.3 Query generated by RacerPro

When RacerPro performs the action `retrieve-with-explanation` it automatically creates a corresponding query that contains all information necessary to add the resulting explanations to the knowledge base. Namely, the query contains information about new individuals and hypothesized assertions for each explanation. RacerPro's command `(add-explanation-assertions :queryid i)` adds explanation number i from the specified query to the current knowledge base.

While this information obviously contains everything needed by RacerPro for the corresponding knowledge base augmentation, the situation might be somewhat different for the queries sent to HCI: since those queries are supposed to provide the user with different choices for interpretations of the document, it is crucial to let the user know what the query actually relates to. Without any enhancements, the information directly provided by RacerPro's queries is rather useless to the user.

If, for instance, a user at HCI was in the process of tagging a video that might be of considerable length and contains the previously described situation somewhere, the user would be asked at some point whether *IND-3* is a construction site or *IND-2* is a construction yard, given that no additional information was presented to him. In this case, it is very likely that the user does not even know where in the video he should look for the corresponding explanation. The presented individual names would not be of any help, since they are rather arbitrary names (at least from the user's point of view) and therefore do not allow any conclusion about their actual references to the video content.

On the other hand, RacerPro's queries contain supplementary information that is needed for the addition of explanations to the knowledge base but which is rather unnecessary for HCI to present the explanation's meaning to the user. Hence, instead of using the queries natively used within RacerPro, it is desirable to use a different query format for the communication to HCI. Details on this query format will be introduced later on in this chapter, but before a suitable query format is discussed, it is necessary to have a closer look at the information available with HCI.

4.2 Approaches of knowledge communication to HCI

HCI will provide the user with the multimedia files that are going to be interpreted. Additionally, all assertions from KDMA and HCI are mutually exchanged between all participating components (and of course, later on, RMI will contribute additional assertions that have been derived during the interpretation process). Hence, it can be safely assumed that at the interpretation process's start HCI has the same set of assertions available that make up RMI's initial ABox. As discussed in the previous section, this information by itself is not yet sufficient to provide a correlation between the actual content and the information on inferred explanations as it is communicated by RacerPro's queries. Different approaches are thinkable to overcome this issue at the client's side:

4.2.1 A dedicated RacerPro instance with unmodified ABox transfers

One approach of making the necessary information available to the user could be the maintenance of a dedicated RacerPro instance within the HCI component. This way, RMI could perform the explanation retrieval and subsequently transfer its complete knowledge base (i. e. a set of ABoxes $A_1, ..., A_n$) to HCI. Hence, all operations on the ABoxes that are required to obtain additional information regarding the explanations could be

encapsulated within HCI. Thus, any queries raised during the interpretation process could remain merely unchanged and directly forwarded to HCI.[1] As a consequence, HCI could enhance the queries locally as needed to meet the discussed requirements regarding the relationship between explanation and content.

While this approach might seem appealing because of its rather simple concept, it would come with to some major drawbacks:

First of all, it is quite obvious that this approach would lead to an undesired communication overhead since the resulting ABoxes produced by RMI would be transferred to the client as a whole although only a fraction of this data is needed in order to specify the derived explanations.

Also, it is highly questionable whether it is really necessary to maintain a dedicated RacerPro instance at the HCI component. Although this might enhance HCI's flexibility of operations, the additional RacerPro instance would increase maintenance efforts and also it might lead to raised computing performance requirements at HCI since supplementary information retrieval operations are to be performed. Hence, the separate installation of RacerPro at the client should be avoided if possible.

Additionaly, there is a practical reason to avoid a RacerPro instance at the HCI component: an implementation based on RacerPro requires profound knowledge in the field of knowledge engineering. The different components of CASAM are being developed by separate teams and the experts on knowledge engineering are gathered at TUHH's Institute for Software Systems which is responsible for RMI's development. Therefore it is advisable to encapsulate all tasks related to RacerPro within the RMI component instead of distributing those tasks among different components and thereby among different developer teams.

4.2.2 A dedicated RacerPro instance with ABox delta transfers

The approach regarding a dedicated RacerPro instance described above can be optimized in order to reduce the mentioned communication overhead. Instead of transferring the ABoxes $A_1, ..., A_n$ without any modification to the client, RMI could use one resulting ABox $A_{ref} = A_1$ as a reference and compute a set of ABox differences $\Delta^+_{ref,2}$, $\Delta^-_{ref,2}$, ..., $\Delta^+_{ref,n}$, $\Delta^-_{ref,n}$. Those differences $\Delta^+_{ref,k}$ and $\Delta^-_{ref,k}$ provide means for the switching between different resulting ABoxes by specifying all assertions that need to be added and subtracted respectively to A_{ref} in order to obtain A_k.

Consequently, instead of transferring n ABoxes from RMI to HCI, only one complete ABox A_{ref} and $2 \cdot (n-1)$ Δs have to be sent. Since in practice the Δs' size is only a small fraction of the ABox's size, the communication overhead can be reduced by a considerable amount.

Except for the communication needs, the same considerations from the previously discussed scenario apply here: the queries to HCI may be sent without any previous enhancements, but a dedicated RacerPro instance is necessary to retrieve additional information. (Actually, in this scenario RacerPro would even be assigned with the additional task of computing the resulting ABoxes from the set of Δs, although there might be options to implement the ABox additions alternatively without using RacerPro).

For further optimization, it might be possible to compute the Δs with respect to the input ABox instead of one of the results. Since this ABox is already present at HCI, no complete ABox would have to be transferred but only a set of $2 \cdot n$ differences. However, while it is possible that different results only exhibit small differences with respect to each other, it might very well be that all of them are of great difference to the input ABox.

[1] As mentioned before, those queries are not perfectly suitable for the intended use. However, since the information required for this scenario is basically a subset of the information provided by RacerPro's queries, minor query changes are neglected here.

As a consequence, waiving of the single complete ABox transfer might lead to Δ's that are larger in size and thus the amount of data transferred overall will be not necessarily reduced.

For instance, see Example 3: before the interpretation process branches for the first time, three assertions have been hypothesized already. If differences were computed with respect to the input ABox, those assertions would have to be transferred in each Δ^+; if one of the resulting ABoxes was used as a reference, those assertions would be transferred only once.

Hence, the best choice of reference cannot be clearly determined on a general basis but is rather dependent on the individual situation. However, since even an optimal choice would only lead to minor improvements with respect to the communication overhead while leaving the other drawbacks untouched, this approach is not followed further.

Additionally, it should be noted that the ABoxes produced by RMI in fact contain more information than actually required for HCI's tasks (for instance, see Figure 4.2: while the *builtFrom*-relation is important for the interpretation process, it is not necessary to make this information available to HCI). Since the Δs contain all information necessary for a reconstruction of RMI's resulting ABoxes, there is obviously more data transferred than necessary and thus there is further potential for the reduction of communication overhead.

4.2.3 A supplementary communication channel to RMI's RacerPro instances

The considerations from the previously discussed scenario lead to a different possible architecture that maintains the same operational principles but overcomes the need for a separate RacerPro instance at the client: since RMI maintains several RacerPro instances anyway with the current knowledge at their disposal, the presence of those instances can be exploited by establishing a supplementary communication channel to them, instead of performing necessary information retrieval tasks locally at the client.

In this scenario, the initial communication needs from RMI to HCI can be reduced to just the set of queries. Any further data that is needed by HCI in order to augment the query to a user-friendly presentation can be obtained via this channel. Since additional properties can be requested in a target-oriented way, the amount of transferred data is significantly reduced and only information essential to HCI's tasks is communicated.

This approach reduces the traffic between RMI and HCI on such a large scale that any potential further reduction on the data's size would not yield to a noticeable performance gain. However, opposed the approaches discussed previously, which were based on a uni-directional communication scheme (if HCI's answer to a query is neglected), this approach is subject to a bi-directional request-response-scheme that might introduce additional delays due to several messages sent back and forth between the components. Even though with modern communication networks it is rather unlikely that those delays probably will have a huge impact on the overall performance, they should be avoided if possible.

4.2.4 Compensating RacerPro with enriched queries

As could be seen from the previous scenario, it is possible to eliminate the necessity for a dedicated RacerPro instance within the HCI component. Pursuing the approach of exploiting knowledge already present on RMI's side leads to a further optimization:

The types of supplementary information required by HCI for the purpose of presenting meaningful choices to the user are not really situation-depended, but rather they are always of the same type and thus can be known beforehand. Namely this information consists of the linking between an explanation and its actual position within the multimedia document. While the actual content of this information will obviously vary for each query, it will

```
1  QuerySetType
2  {
3     QueryType   id=1   importance=1
4     {
5        LogicalCompoundType   logicalOperatorType=OR
6        {
7           AssertionType
8           {
9              subject="http://www.sts.tu-harburg.de/casam/abox.owl#IND-3",
10             relation="http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
11             object="http://www.sts.tu-harburg.de/edo.owl#ConstructionSite",
12             actiontype=ADD
13             certainty=0.5
14          }
15          AssertionType
16          {
17             subject="http://www.sts.tu-harburg.de/casam/abox.owl#IND-2",
18             relation="http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
19             object="http://www.sts.tu-harburg.de/edo.owl#ConstructionYard",
20             actiontype=ADD
21             certainty=0.5
22          }
23       }
24    }
25 }
```

Figure 4.3: Sample query created by RMI's current prototype

always be obtained from the segment locators corresponding to the segments involved in the current interpretation.

Thus, as soon as a query is generated by RacerPro, RMI is able to know where to find the additional properties that will be needed by HCI. Hence, instead of providing these properties on demand, it is possible to enrich the queries with this information before they are sent to HCI. As a result, HCI can directly exploit the query's content in order to present the relevant choices to the user and thus the need for any additional ABox operations on the client side dissolves.

Also, since the data transferred between RMI and HCI is reduced to the essentials, this approach is highly efficient with regard to communication aspects as well.

Consequently, the remainder of this work is based on this scenario.

4.3 Queries: syntax and semantics

4.3.1 Queries generated by the current prototype

As an introduction to the queries created by RMI, Figure 4.3 depicts the query corresponding to the previously discussed construction example as it is created by RMI's current prototype.

The query's frame

For the creation of queries, CASAM has a data type *QuerySetType* (line 1) in which the query is embedded. The query itself is of type *QueryType* (line 3) with the properties *id* and *importance*. The id is a unique number that is created by RMI for each query in order to provide means for the identification of different queries and hence a correlation between certain interpretation steps and corresponding answers from HCI. The importance property enables RMI to assign differentiated priorities to different queries depending on the information gain RMI might obtain by a query's answer. HCI can exploit the importance value to alter the order in which queries are presented to the user. Details on query priorities will be discussed later on in this chapter.

Different assertions within a query are bundled via the *LogicalCompoundType* with the property *logicalOperatorType* (line 5). Basically, the locialOperatorType can take two different values: *OR* signals that a choice of different alternatives is presented, while *AND* denotes an aggregation of multiple assertions into a single explanation.

Assertion essentials

Each assertion within a logical compound is exhibited by an *AssertionType* (e. g. lines 7-14) that contains detailed information about each assertion, namely a subject, an object, and a relation between those; the subject provides an individual's name as it is used within the knowledge base, the object and its relation will provide an additional property of this individual. Although future implementations of RMI will make use of different query types (namely *Concept Assertion Queries*, *Role Assertion Queries*, and *Same-As Queries*, see [TUH] for details), currently only queries of the type *Concept Assertion Query* are being generated. For this query type, the assertion's object always denotes a class from edo and the assertion's relation declares the specified individual as a type of this class. Consequently, all examples presented in this work only yield to this query type. However, the principles discussed for this query type might be applied to the other types as well.

Assertion actiontypes

In addition to the assertion's elementary properties the assertion type contains a value *actiontype*. This value specifies how the provided assertion should be treated: the value *ADD* signals that this assertion should be added to the current knowledge base. The contrary value *DEL* would instruct HCI to remove the specified assertion from the current knowledge base. Thus, those actiontypes enable HCI to alter its current knowledge base if some kind of delta-transference approach is being pursued.

Since the approach selected for this work in fact does not require HCI to perform operations on the knowledge base at all, this parameter is not being used. Since the assertions contained within the queries are results of RMI's interpretation process (and thus represent additional knowledge compared to HCI's currently available knowledge) it might be assumed (if this property's existence is desired) that those queries always contain the operator *ADD*, although there is actually no addition performed at HCI.

Assertion certainties

The final supplement to each assertion is a *certainty* property. This property may take values in the range from 0 to 1 and represents how certain RMI is about a particular explanation. Currently, this value is subject to an equipartition among all possible assertions (for n explanations, the certainty value for each explanation yields to $1/n$). As can be seen directly from those values, RMI currently is not able to assign differentiated certainties to different explanations and thus every explanation yields to the same certainty.

Obviously, an equipartition in this context is not able to provide any information gain at all. However, it should be noted that the currently used equipartition is only a placeholder for later developments: as already announced in [GMN$^+$09], future implementations of RMI will exhibit a probalistic approach for an explanation's scoring. Once this approach is being used, it is rather likely that different certainties evolve for different explanations and thus this property might contain quite meaningful information.

Even though this property might contain meaningful information in the future, it will still not be necessary to transmit this information to HCI: if the explanation's certainty has an impact on RMI's priority to obtain the query's answer, this can be incorporated into the importance value that is being transmitted anyway. Beyond the query's priority

there is no need to share this information at all, but it might be solely used to augment the query's presentation to the user.

Hence, the subsequent discussions will neglect this property as well.

4.3.2 Enriched queries

While the query's frame may remain unmodified for the intended purpose, the content of *AssertionsType* is subject to some changes in order to fit the knowledge communication approach selected for this work:

First of all, the supplementary properties *actiontype* and *certainty* are discarded because they do not contribute to any information gain, as described in the previous section. Note that their removal is not mandatory; since they do not cause any interference to the query process it is also possible to keep them within the query. Consequently, their removal is due to simplicity considerations rather than a real necessity.

In principle, if an assertion's subject or object contains an individual name, this might be removed as well, since the name used internally by RMI does not shed any light on the query's content either. However, in order to allow for an easy following of the presented examples, those names are kept within the queries at least for the scope of this work.

To enrich the query with information required by HCI, the supplementary information *SegmentLocator* is added to the *AssertionType*, as depicted in Figure 4.4. Each *SegmentLocator* is specified via its *type* property and it contains the actual positioning information corresponding to the hypothesized individual.

To obtain this positioning information, all *builtFrom* relations of the hypothesized individual are traversed recursively until all input assertions involved in the interpretation are identified. For those assertions corresponding segments and therefore segment locators can be obtained. As a simple approach for audio and video modalities, all segment locators involved are aggregated and the lowest *hasStart* and the highest *hasEnd* values are being used. Although this is a fairly simple approach that might be subject to major improvements, it should be sufficient to allow for an easy identification of the corresponding position within the multimedia document.

While it might appear simpler to add the *hasStart* and *hasEnd* properties directly to the *AssertionType* at first glance, the approach of nesting the actual properties into the type *SegmentLocator* provides more flexibility: note that not all media modalities use the same type of segment locators. For instance, mco provides the definition of a *BoundingBox* for the image modality with information about its four corner coordinates. Hence, via specifying the type of locator within the query, the types of positioning information nested within may be adjusted to the current needs.

Due to this supplementary data, HCI now has all required information at its disposal to achieve a clear marking of the explanation's position within the multimedia documents.

4.3.3 High-level queries

The queries discussed so far only adhere to single interpretation steps. Consequently, this approach would force the user to answer all queries on a certain interpretation path in a top-down fashion. For instance, consider the whole result set of the first interpretation example: since the scenario is rather simple, it is quite likely that the user would be able to decide instantly whether the scene depicts a bridge building or a dam building (assuming that one of these interpretations is true). However, since the final interpretation result relies on two preceding hypotheses, it is not possible to decide on one of those results without confirming all involved hypotheses first. Thus, the user is forced to clarify both preceding interpretations (i. e. reply to query 1 and query 2) before being allowed to reply to query 3.

Though the question as to whether or not it is actually the best solution to prompt the user for replies on the highest-level queries first is subject to further discussion, it is obvious that this option might lead to a significantly more convenient way of answering queries in certain situations. Thus, the queries should at least provide the means of enabling this option.

Hence, for high-level interpretations, all hypothesized explanations involved should be aggregated into a single query that may be answered by the user instantly. The query frame introduced previously already provides everything necessary for this aggregation: since a high-level interpretation consists of a conjunction of hypothesized assertions, those assertions can be attached to one query by using *logicalOperatorType=AND*. Figure 4.5 depicts a first approach to such an aggregated query: it presents a choice of different sets of hypothesized assertions, namely the choice between the interpretations *(Construction AND ConstructionSite) OR (Destruction AND ConstructionSite)*.

Although this approach fulfills the requirements of a high-level query, it can easily be seen that the query contains redundant information, and therefore it is not an optimal choice. In addition to an unnecessary blow-up of the query's size, a repetition of hypotheses common to all choices will prevent an easy identification of the actual differences between the presented options. By exploitation of logical equivalences, the query can be reformulated in such a way, that the query's emphasis then lies on the actual differences followed by a trail of common hypotheses. This yields to the query *(Construction OR Destruction) AND ConstructionSite*.

Analogously, the last interpretation step from the first examples leads to the optimized query *(DamBuilding OR Bridgebuilding) AND Construction AND ConstructionSite*, as it is depicted in Figure 4.6.

Note that those high-level queries enable the user to select each of the interpretation results through answering a single query independently of the result's interpretation level.

```
1  QuerySetType
2  {
3     QueryType   id=1   importance=1
4     {
5        LogicalCompoundType   logicalOperatorType=OR
6        {
7           AssertionType
8           {
9              subject="http://www.sts.tu-harburg.de/casam/abox.owl#IND-3",
10             relation="http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
11             object="http://www.sts.tu-harburg.de/edo.owl#ConstructionSite",
12             SegmentLocator   type=VideoLocator
13             {
14                hasStart="00:00:10:000",
15                hasEnd="00:00:42:000"
16             }
17          }
18          AssertionType
19          {
20             subject="http://www.sts.tu-harburg.de/casam/abox.owl#IND-2",
21             relation="http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
22             object="http://www.sts.tu-harburg.de/edo.owl#ConstructionYard",
23             SegmentLocator   type=VideoLocator
24             {
25                hasStart="00:00:10:000",
26                hasEnd="00:00:42:000"
27             }
28          }
29       }
30    }
31 }
```

Figure 4.4: Example for an enriched query (query 1 from the first example)

Opposed to this highly convenient way of query-answering, the previous approach of only presenting the current interpretation's alternative forced the user to answer one query per interpretation level.

4.4 Prioritizing queries

As already mentioned before, RMI should be able to assign differentiated importance values to each query in order to alter the queries' presentation order. Before discussing details on possible priority criteria, it is necessary to examine RMI's communication capabilities (and more importantly: possible actions upon received messages).

4.4.1 Communication during the interpretation process

From an external point of view, RMI's currently implemented interpretation process (as described in the previous chapters) can be seen as an atomic operation: at some point the process is initiated based upon a certain set of inputs and successively RMI performs all interpretation steps possible based on current input and rules. The process will terminate only if either no further rules can be applied to the current knowledge base or else if some other kind of termination condition (see Section 2.4.3) has been met. Before meeting one of these termination conditions, RMI's current prototype will not react upon any external events (i. e. the process is non-interruptable).

In the context of query-processing this is a rather undesirable behavior: while RMI is still in the process of retrieving high-level interpretations, it is very well possible that HCI already returned replies to some queries that have arisen earlier. Obviously, those query replies contain additional information that would enable RMI to eliminate a subset of explanations. By looking at the interpretation process as a tree of several interpretation paths (as presented in the previous chapter), each query answer could allow RMI to neglect the pursuance of all but one of the corresponding node's subtrees. For instance, consider the third example's interpretation process as depicted in Figure 3.12: imagine that RMI has just performed the interpretation step *Explain: building1, doorSlam2* (with the corresponding rule 3) and in the meantime received an answer to *Query4*, saying that a *CarEntry* is depicted. If RMI was able to evaluate this information while still in the process of interpreting, the subtree containing *CarExit* could be eliminated. Thus, two interpretation steps would have been eliminated and therefore two queries to HCI could have been saved.

Since the tree's shape and depth are highly dependent on the respective input assertions as well as the applicable rules, the number of interpretation steps that might be saved by such a tree-cutting function will vary on a large scale for different scenarios. Thus, it is not possible to quantify any potential performance gain in a general way. However, under the assumption (that should hold up in practice) that most input observations can be explained via some rule and that a majority of rules will yield to different possible explanations, it becomes obvious that the required computation time is subject to an exponential growth with an increasing number of input assertions.

Consequently, RMI should be expanded with a functionality to react upon incoming query answers while still interpreting (i. e. the interpretation process should become interruptable) in order to reduce computation requirements and to spare the user the evaluation of unnecessary queries. The following section will present criteria for priority assignments if an interruptable process is being used.

```
QuerySetType
{
   QueryType   id=2  importance=1
   {
      LogicalCompoundType   logicalOperatorType=OR
      {
         LogicalCompoundType   logicalOperatorType=AND
         {
            AssertionType
            {
               subject="http://www.sts.tu-harburg.de/casam/abox.owl#IND-5",
               relation="http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
               object="http://www.casam-project.eu/edo.owl#Construction",
               SegmentLocator   type=VideoLocator
               {
                  hasStart="00:00:10:000",
                  hasEnd="00:00:42:000"
               }
            }
            AssertionType
            {
               subject="http://www.sts.tu-harburg.de/casam/abox.owl#IND-3",
               relation="http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
               object="http://www.casam-project.eu/edo.owl#ConstructionSite",
               SegmentLocator   type=VideoLocator
               {
                  hasStart="00:00:10:000",
                  hasEnd="00:00:42:000"
               }
            }
         }
         LogicalCompoundType   logicalOperatorType=AND
         {
            AssertionType
            {
               subject="http://www.sts.tu-harburg.de/casam/abox.owl#IND-4",
               relation="http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
               object="http://www.casam-project.eu/edo.owl#Destruction",
               SegmentLocator   type=VideoLocator
               {
                  hasStart="00:00:10:000",
                  hasEnd="00:00:42:000"
               }
            }
            AssertionType
            {
               subject="http://www.sts.tu-harburg.de/casam/abox.owl#IND-3",
               relation="http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
               object="http://www.casam-project.eu/edo.owl#ConstructionSite",
               SegmentLocator   type=VideoLocator
               {
                  hasStart="00:00:10:000",
                  hasEnd="00:00:42:000"
               }
            }
         }
      }
   }
}
```

Figure 4.5: Second-level query (corresponding to the first example's query 2, not optimized)

```
QuerySetType
{
   QueryType   id=3  importance=1
   {
      LogicalCompoundType   logicalOperatorType=AND
      {
         LogicalCompoundType   logicalOperatorType=OR
         {
            AssertionType
            {
               subject="http://www.sts.tu-harburg.de/casam/abox.owl#IND-7",
               relation="http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
               object="http://www.casam-project.eu/edo.owl#DamBuilding",
               SegmentLocator   type=VideoLocator
               {
                  hasStart="00:00:10:000",
                  hasEnd="00:00:42:000"
               }
            }
            AssertionType
            {
               subject="http://www.sts.tu-harburg.de/casam/abox.owl#IND-6",
               relation="http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
               object="http://www.casam-project.eu/edo.owl#BridgeBuilding",
               SegmentLocator   type=VideoLocator
               {
                  hasStart="00:00:10:000",
                  hasEnd="00:00:42:000"
               }

            }
         }
         AssertionType
         {
            subject="http://www.sts.tu-harburg.de/casam/abox.owl#IND-5",
            relation="http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
            object="http://www.casam-project.eu/edo.owl#Construction",
            SegmentLocator   type=VideoLocator
            {
               hasStart="00:00:10:000",
               hasEnd="00:00:42:000"
            }
         }
         AssertionType
         {
            subject="http://www.sts.tu-harburg.de/casam/abox.owl#IND-3",
            relation="http://www.w3.org/1999/02/22-rdf-syntax-ns#type",
            object="http://www.casam-project.eu/edo.owl#ConstructionSite",
            SegmentLocator   type=VideoLocator
            {
               hasStart="00:00:10:000",
               hasEnd="00:00:42:000"
            }
         }
      }
   }
}
```

Figure 4.6: Third-level query (corresponding to the first example's query 3, optimized)

4.4.2 Runtime priorities

Basic priorities

The considerations from the previous section immediately lead to a priority scheme that should be used while the interpretation is still running: since a query's reply has the purpose of cutting subtrees, it is obvious that a query's importance should be directly correlated to the subtree's size that could be cut by its reply. Of course it is not possible to determine each subtree's actual size before its corresponding interpretations have been carried out entirely, but it can be safely assumed that a subtree's potential size shrinks with an increasing distance between its parent node and the root node. In other words, the higher the interpretation's level, the smaller its subsequent number of interpretations.

Hence, an interpretation's level is used to determine its corresponding query's importance. Assuming a priority range from 0 to 100 (with 100 being of highest importance), an importance value of 90 would be assigned to first-level-queries, 80 to second-level-queries, and so on. This scheme intentionally leaves a gap between the most important level and the highest possible importance value to allow for further refinements, as discussed below. Note that only the difference between different values is important; the actual value range and assignment are selected rather arbitrary for this work and might easily be altered to fit practical requirements. In order to practically determine an explanation's level, one could traverse all of its *builtFrom* relations: the length of the longest chain of *builtFrom* relations, from the current explanation's individual to an input assertion, will determine the explanation's level.

Refined priorities

For a further refinement of priorities between different queries from the same level the number of alternatives within each query can be analyzed: since each choice within a query represents a branch in the interpretation tree (and thus leads to a separate subtree), a reply to a query with several choices will lead to several subtrees being cut. Thus, the importance of a query increases with the number of choices it represents. In order to reflect this in the query's importance, its importance value is incremented by one for each alternative contained within, after assigning the level-priorities to each query.

This yields to the following equation for the determination of a query's importance in a top-down way:

$$I_{TD}(q) = (10 - L(q)) \cdot 10 + \#a \tag{4.1}$$

$$\begin{aligned} I_{TD}(q) &:= \text{Importance of query } q \\ L(q) &:= \text{Interpretation level of } q \\ \#a &:= \text{Number of alternatives contained within } q \end{aligned}$$

Note that this equation imposes certain restrictions upon the interpretation results: no interpretation step is allowed to yield to more than nine different explanations and the interpretation's depth is limited to ten levels. While exceeding those limits would not have severe consequences on the interpretation itself, some importance values would be outside of the defined value range and thus the priorities would become useless.[2] However, if realistic scenarios will prove these boundaries to be too tight, the equation can be adapted quite easily.

[2] As long all interpretations adhere the restrictions, the actual value range is in fact from 1 to 99, but of course that does not create any problems.

Therefore, the replacement of static values in the previous equation leads to the general expression for the calculation of importance values as exhibited in equation (4.2): instead of using fixed values, three variable parameters are introduced that allow an easy adaption to different requirements: d denotes the interpretation's maximum depth (i. e. the highest-level explanation that may occur during the explanation retrieval process) and w denotes the maximum width (i. e. the maximum number of alternatives) for any interpretation level. Note that the value $(w+1)$ is being used for calculation to ensure that the importance value for queries with a maximum number of alternatives does not stretch out over the next-lower level. By usage of these parameters, the lowest possible importance value 1 occurs for a highest-level query with a single assertion contained within, while a highest-level query with a maximum number of alternatives leads to the highest possible importance value of $(d-1)(w+1)+w$. To ensure that those extreme values stay within the desired boundaries of importance values, the additional scaling constant s is introduced. Consequently, it holds that $I_{TD}(q) \in \left[\frac{1}{s}, \frac{(d-1)(w+1)+w)}{s}\right] \ \forall\ q$ and thereby an easy adjustment to the desired range is possible by an alteration of s.

$$I_{TD}(q) = \frac{1}{s}[(d - L(q)) \cdot (w + 1) + \#a] \tag{4.2}$$

$$\begin{aligned} d &:= \text{Maximum interpretation depth} \\ L(q) &:= \text{Maximum interpretation width (for each level)} \\ s &:= \text{Scaling constant} \end{aligned}$$

Unary queries

Now it is time to return to the question raised during the discussion of the second interpretation example (see page 22): should a unique interpretation result (i. e. without ambiguities) produce a query (called *unary query* in the following)?

It is evident that a positive reply to a unary query does not contribute to a valuable information gain of any kind: although this answer would confirm a hypothesis previously made by RMI, the interpretation process does not have any alternatives available and therefore the interpretation's course will remain unaltered, regardless of its confirmation status. Thus, it is needless to prompt the user for positive answers to a unary query.

However, even though no reply to a unary query is able to alter an interpretation's direction, negative answers to those queries can still be exploited to achieve a simplification of the process. To illustrate this, the interpretation from the third example (Figure 3.12) is considered: RMI started off with three interpretation steps that yielded to unary queries before branching into different paths. All subsequent steps on higher levels entail at least one of the hypotheses generated by these first three steps. Now it is assumed that unary queries have been sent to HCI: this enables the user to follow the process in real time. If he agrees to the hypotheses generated by RMI, he is not required to take any actions but he can wait until further queries arrive. However, if he is able to identify a flaw in one of the hypothesis, he is able to reject it immediately by submitting a negative answer to the corresponding unary query. For instance, it is presumed that RMI has just finished the first three interpretation steps from example 3 and sent the according unary queries to HCI. The user did not agree to a *CarRide* being detected in the input data and therefore he supplies a negative answer to query 1. RMI is able to immediately cut out large portions of the interpretation tree. Hence, the possible results are limited to ABox5 and ABox9 while the number of queries is reduced to five (namely the aforementioned unary queries plus query 5 and query 8). The same situation without sending unary queries to HCI would have resulted in a total of seven queries (with five of them being completely needless since they incorporate the hypothesis *CarRide*).

Hence, next to reducing computational requirements, the supplement of unary queries embodies the potential of significantly reducing the total amount of queries a user has to handle. Since their replies only yield to a reduced knowledge gain (opposed to other queries), they should be of lower priority. Since the importance function described above is influenced by the number of choices contained within a query, unary queries automatically get the lowest importance value of all queries within the corresponding interpretation level. If desired, their importance could be lowered further by assigning a negative weight to unary choices.

Further improvement

Further pursuing the presented idea of prioritizing queries during runtime could lead to an additional improvement: since the query importance is supposed to allow for a cutting of unprocessed subtrees, a query's importance evidently fades once all of its subtrees have reached leaves (i. e. all paths have been followed until no more rules are applicable). Accordingly, in this state a query's reply would again lead to no alteration of the interpretation process and thus its corresponding importance could be drastically lowered a posteriori. However, this idea would lead to a constant alteration of the queries' importance values and therefore lead to a significant amount of traffic between RMI and HCI. Since the priority scheme introduced so far should assure a rather expedient use of priorities, a weighting of costs and benefits leads to a waiving of this improvement.

4.4.3 A posteriori priorities

While the derivation of runtime priorities was fairly straightforward, the importance requirements are harder to identify once the interpretation process has terminated.

First of all, the arguments provided for the use of unary queries do not hold up any longer: since their main goal was a possible early termination of the interpretation process, they completely lose their purpose in the after-runtime phase; instead of enabling a reduction of the query's quantity, they just become a hassle to the user (the information obtained by the rejection of a unary query can also be obtained through replies of higher-level queries). Therefore, unary queries should be discarded a posteriori (or alternatively, their importance value should be set to 0).

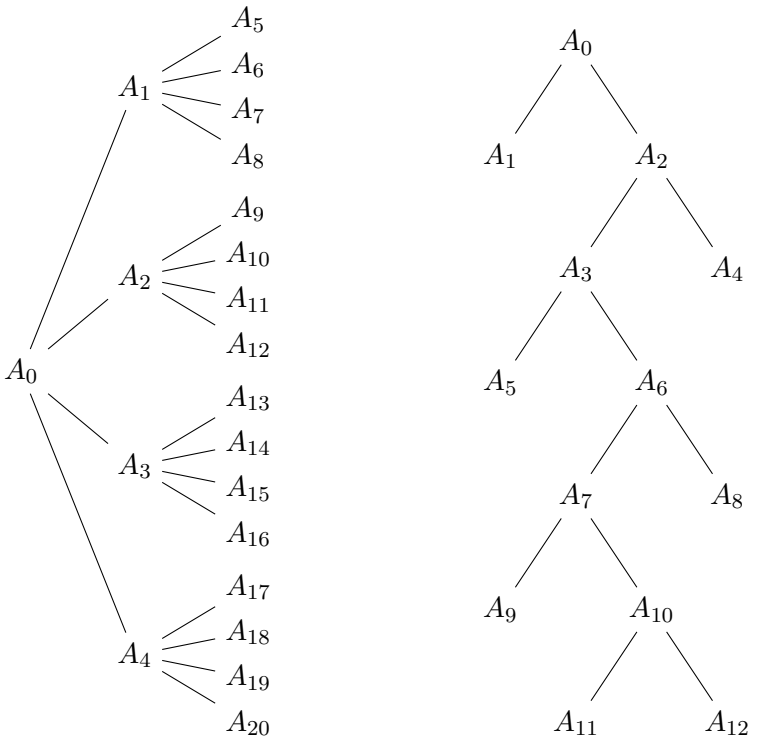
Figure 4.7 shows the schematic interpretation courses for different (imaginary) examples: each node depicts an ABox A_k, subsequent nodes depict possible explanations of an ABox. Since only the tree's shape is of interest for this discussion, interpretation steps are not shown explicitly.

(a) depicts only a few interpretation steps, each resulting in multiple explanations. Hence, the number of possible solutions is quickly growing while the interpretation's levels stay fairly shallow. (b) depicts the opposite situation where only one path leads to high-level interpretations while all alternative paths turn out to be dead ends.[3] Even though it has to be admitted that those artificial examples denote extreme cases and thereby do not necessarily reflect realistic scenarios, they still illustrate the need for different priority needs quite well.

Priorities in width-oriented interpretations

Examining the tree's shape in (a) leads to an intuitive scheme on the queries' priorities: it is evident that a top-down approach will require the user to reply to exactly two queries (and therefore consider eight choices altogether) independently from the actual result. If

[3] Of course, this does not conclude that those leaves are wrong explanations, it simply says that they cannot be interpreted any further.



(a) width-oriented interpretation

(b) depth-oriented interpretation

Figure 4.7: Schematic interpretation trees

a bottom-up approach was used instead (i. e. presenting high-level queries first), the user would be able to select the right result by answering a single query. However, there are four different second-level queries (all with the same priorities) and each of them contains four different choices. Hence, there would be a total of 16 different choices a user would have to consider before deciding on the right one and thus even though one query reply was saved, this approach would require more effort from the user.

Consequently, in this scenario the top-down approach from the runtime situation can be adopted without any modification and therefore equation (4.2) can be used to determine a query's importance value.

Priorities in depth-oriented interpretations

The example depicted in (b) is similar to the construction example from the previous chapter: although each step yields to two different explanations, only a single path leads to higher-level interpretations. Since two high-level interpretations (in A_{11} and A_{12}) seem to perfectly fit the observations, this scenario suggests that those are the likely explanations while all other explanations yield to early termination.

The top-down approach that has been used so far leads to an inconveniently large amount of queries that a user would have to answer before getting to the final result: under the assumption that indeed either A_{11} or A_{12} contains the right solution, the user would have to answer six queries, each containing two choices. On the other hand, if a bottom-up approach was used, the reply to a single query with two choices would be sufficient to clarify the results.

If a scenario like this contains a few queries on the highest interpretation level, the same refinements that have been discussed before can be applied to this situation: the more

choices a query contains, the higher its importance value becomes. Although in this case, none of the queries is really more important than other queries on the same level, an ordering by the number of choices will increase the chances that the desired solution will be found in one of the first queries.

This leads to a slightly modified importance function incorporating bottom-up priorities:

$$I_{BU}(q) = \frac{1}{s}[L(q) \cdot (w+1) + \#a] \tag{4.3}$$

The resulting range of importance values leads to a minimum value for a first-level query with a single assertion, while the maximum value occurs for a query on level d with w assertions. Hence, $I_{BU}(q) \in \left[\frac{w+2}{s}, \frac{d(w+1)+w)}{s}\right] \ \forall \ q$. Note that even though equation (4.3) does not incorporate the parameter d, it is still necessary to determine its value in order to ensure appropriate boundaries.

Choosing the suitable approach

Now that two different priority assignment schemes are available, after completing the interpretation process it is necessary to decide on one approach. Unfortunately, realistic scenarios are unlikely to exhibit one of the tree's shapes as clear as depicted in Figure 4.7. In fact, practical scenarios are likely to result in several explanations distributed among different interpretation levels. For instance, a single interpretation could be added to Figure 4.7a as a subsequence to A_5. Assuming that high-level interpretations are usually preferred, the bottom-up approach would correctly assign the highest priority to this new interpretation. However, if this interpretation turns out to be wrong, the following queries created by the bottom-up approach would lead to the problems discussed previously.

In order to adapt the priority assignments to mixed scenarios, it is proposed to analyze each interpretation's level (starting with the highest level) with regard to its depth and width (i. e. level and number of queries within this level). If the Ration $R = \frac{d}{w}$ between depth and width is larger than 1 (i. e. a scenario similar to 4.7b), all queries should get importance values according to the bottom-up approach.

Once the procedure arrives at a level where $R > 1$ does not hold, the remaining levels should be prioritized according to the top-down approach. This hybrid approach allows an emphasis on few high-level explanations, as needed for a depth-oriented interpretation, but at the same time enables a quick tree-traversal if an initial portion of the tree is wider than deeper.

Note that an additional scaling of importance values is necessary once the procedure has switched to top-down priorities, or else the following priorities might interfere with some of the previously assigned values. The easiest way of achieving this is to assign the number of the last bottom-up priority level L_{BU} to d in the I_{TD}-equation leading to the equation:

$$I_{TD}(q) = \frac{1}{s}[(L_{BU} - L(q)) \cdot (w+1) + \#a] \tag{4.4}$$

Hence, the procedure would start with the highest level, assigning I_{BU} values as long as $R < 1$ holds. If this condition does not hold for a certain level (i. e. the level $L_{BU} - 1$), the remaining queries get I_{TD} values.

Further optimization

The bottom-up approach can be further optimized for an efficient query answering: as described in Section 4.3.3, high-level queries should be created in such a way that the different choices are presented in the query's beginning followed by a "tail" of common assertions ordered by their interpretation level. If such a query is presented to the user,

he is supposed to confirm one of the choices or alternatively decline all of them. If in fact none of the choices presented turns out to be the right one, the mechanisms described so far would only allow for a complete refusal of all alternatives. Consequently, the nextmost important query will be presented to the user.

However, if a query is declined completely it is highly probable that part of the query's tail contains false assertions and following queries will most certainly contain a subset of the same tail. Since this tail is presented to the user anyways, he would be able to recognize the false assertions immediately after examining the first query. A complete rejection of this query would simply lead to a presentation of the next query that might contain the same false assertions, since there is no mechanism that enables the user to identify a false assertion. Because the tail is ordered by the corresponding assertions' levels, pointing out the first wrong assertion would allow a direct "jump" to the corresponding subtree and therefore simplify the query answering process significantly. Therefore, the query answering process should be augmented with an option to point out the first error in a list of assertions.

Also, the priority of high-level queries could be further refined by applying a scoring function similar to the one presented in Section 2.4.1: it can be assumed that a high-level interpretation is the better, the more ground observations are entailed. Thus, the *consilience* value of an explanation can be added to the corresponding I_{BU} value.

The *simplicity* value should not be used for the importance adjustment of high level queries; since this value penalizes a high number of hypothesized assertions, it would lead to a preference of low-level interpretations (the number of hypotheses obviously has to increase with a rising interpretation level).

5 Summary and outlook

Goal of this work was the analysis of communication requirements between RMI and HCI, as well as the proposal of a suitable query design. Through a comprehensive discussion of different sample scenarios, key properties of the queries were identified. Based upon those findings, a feasible format for the generation of queries has been developed with the intention of enabling a lean knowledge communication, as well as providing an ergonomic way of query handling for the user.

The queries' structure proposed in this work meets both of these requirements: by enriching the queries with supplementary locating information, all relevant knowledge can be communicated to the user interface with very little communication overhead. By assigning differentiated priorities to each query, the user is able to handle most important queries first, so that the right explanation can be identified quickly. Next to the user's convenience, prioritized queries with runtime processing may benefit the interpretation's speed, also, by an early elimination of possible explanations.

Throughout this work, several potential improvements for future work have been identified:

A slight alteration of the interpretation algorithm's order of explanation retrieval, as described in Section 3.3.2, can yield to a significant reduction of interpretation steps. This accelerates the interpretation process and, at the same time, reduces the amount of queries sent to HCI.

Also, this work encompasses a rather simple approach of enriching queries with positioning information. A more sophisticated approach of determining this information for hypothesized individuals could lead to a more precise presentation at the user interface. Additionally, an integretation of positioning information directly into RMI's knowledge base might be helpful: next to providing input to the queries, this information could be exploited in later interpretation steps, given that the rule definitions are modified accordingly.

Finally, as described in the previous chapter, the proposed prioritizing scheme leaves room for further refinements, for instance by incorporating the idea of a scoring function. The proposed scheme should prove to be sufficient in future implementations, but if practical tests exhibit the need for further refinements, these can be added to the presented importance functions easily.

Bibliography

[BCM+07] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2007.

[CAS] CASAM. Approach. `http://www.casam-project.eu/?Page=approach`.

[GMN+09] Oliver Gries, Ralf Möller, Anahita Nafissi, Maurice Rosenfeld, Kamil Sokolksi, and Michael Wessel. *Basic reasoning engine: Report on optimization techniques for first-order probabilistic reasoning*, 2009. CASAM Project Deliverable D3.2.

[Kay] Atila Kaya. *A Logic-Based Approach to Multimedia Interpretation*. Dissertation yet to be published.

[Tha78] Paul R. Thagard. The best explanation: Criteria for theory choice. *The Journal of Philosophy*, 75(2), 1978.

[TUH] TUHH, Institute for Software, Technology and Systems (STS). *CASAM - Query Generation*. presented at CASAM's Lisbon Meeting in October 2009.

List of Figures

List of Tables

Supplementary CD

The CD contains an electronic version of this work, slides of the corresponding presentation, and data for the examples presented in Chapter 3. For further information on the CD's content, see the file *readme.txt* in its root directory.